后端开发必备知识

说明：

本手册目标人群为初级、中级工程师，可用于新员工入职指导和培训。

内容包含互联网后端开发人员通用的必备知识。建议后端开发人手一册。

本手册内容需要及时更新，建议一年修订一次。

1 高可用

1.0 综述（梁海龙）

高可用指的是系统在各种情况下要能继续工作满足用户使用。在互联网行业是保证用户体验的一个重要特性。软件应用，首先应该保证可用，其次才是易用、好用。高可用是互联网产品的生命基石。

要做到高可用，第一点应该为**代码健壮**。代码如果不能在各种输入情况下保证容错，甚至直接崩溃，是无法做到高可用的。

第二点应该是**自动切**。当硬件网络或者其他外部环境故障时，有冗余服务可用。并且在调用方有自动重试切流策略，能及时使用冗余服务。这意味着在 APP、接入层、业务层、数据层均必须有相应设计。

第三点是在生产系统的**故障应急响应**。当故障发生时，如何快速发现，快速定位，快速恢复。在一些可以预见的故障类型，是否可以有自动恢复策略。当无法避免损失的情况发生时，有无隔离策略避免损失扩大，有无降级策略可以舍卒保车。

当不得不人工介入时，是否有紧急预案能快速恢复系统。

第四点是**生产流程的隐患控制**（PS：严格来说这点其实包含了第一点编码、第二点设计，只是因为编码和设计的重要性单独列为第一第二）。整个生产流程包括需求分析、设计、编码、测试、部署发布等环节。各个环节均可能带来各种隐患影响高可用性。比如说设计有无做评审？编码有无做 CodeReview？有无小步迭代？有无灰度发布？有无评估变更影响？监控指标是否有效？有无一键回滚？

1.1 代码容错（向炼）

1.1.1 参数处理

不相信输入，防止被输入搞死，是高可用程序员的基本素质。

输入：必须做有效性检验，包括类型和取值范围，未通过检验的需返回参数错误的错误码，如有特殊情况（老版本客户端导致），只能放行该种特定情况的请求，并同时需对该种错误进行监控；

Ø 公有参数

appid、clientver、serverid、clienttime、servertime、mid、signature、userid。

Ø 业务参数

重定向参数、page size 参数、慢查询参数等。

按照约定输出，不把下游搞死，是程序员做人的基本道德。

输出：需要严格遵守接口文档规定的范围，不能擅自增加输出值，如需修改，需对上下游进行回归测试。

Ø HTTP 码

严格遵守 RFC 文档规定。

注：6xx 属于自定义错误码，目前已使用 609（限流），同时自定义错误码不能随便定义，需申请才能使用。

Ø 错误码

在 HTTP 码之外，为了有利于问题定位，在返回的 Json 中含 error_code 字段。平台监控支持该字段统计。例如：

```
{"status":1,"error_code":0,"data":null}
```

错误码定义请参考平台监控文档：http://10.13.0.34/mediawiki/index.php?title=%E9%94%99%E8%AF%AF%E7%A0%81%E8%A7%84%E8%8C%83

1.1.2 失败处理

在调用库函数或者外部请求的时候，请确保对错误分支做了妥当处理——不崩溃，也不坑队友。

在失败处理上，逻辑的正确性，胜过于代码的整齐简洁。所有语言的异常处理流程都试图让代码简洁，然而却没有谁能做到完美。

◆ 超时设置

调用端的超时一定要大于被调用端超时。

Ø 连接超时时间

同机房：单次 100ms；调用方可以选择重试 2 次。

跨机房：单次 200ms；目前南北机房 RTT 接近 50ms，考虑到业务南北互相调用的情况，且存在丢包或服务处理过来情况。调用方可以选择重试 2 次。

Ø 收发超时时间

大部分接口收发数据在一个 MSS 内，内容大于 4k 的需要走压缩方式。单次超时时间设置：服务 99line 时间 + 网络时间 + 固定时间（为异常预留），一般接口网络时间一般预留 100ms，固定时间预留 500ms，其他的需要根据具体情况而定。

◆ 5xx 返回

HTTP 中的错误码 5xx，代表一种需要重试的错误码，因此调用端需要重试的请求都需要返回 5xx，并同时返回业务错误码（方便监控和排查问题）。默认 ACK、公共接入和 RPC 会对返回 5xx 的结果进行重试。

需注意：返回 5xx 可能会导致流量放大明显（超过 2 倍），可能会对服务产生大的冲击，因此需要合理的对流量进行疏导，设置合理的重试次数，同时业务需要做好服务自我保护功能。

1.1.3 异常处理（包括溢出）

1. **影响：**

部分框架有未捕获的异常会导致不能优雅退出，如 OpenResty 中 Lua 代码抛出未捕获的异常后，Reload 后原有进程不能自动退出。部分语言会由于异常未捕获导致 coredump。非预期结果异常会导致最终结果错误，如使用方未对结果进行严格检验，轻则影响用户体验，重则金钱损失严重。

2. **范围：**

常见的异常包括，语法错误（非编译语言）、非预期结果和运行错误。对 Nil、

NULL 和 0 操作、类型不匹配、非预期数据等。

数值运算时需要关注类型边界，以免溢出导致错误。推荐对输入参数进行范围检验，输出结果做预期检测。

3. **注意事项：**

不要对整段代码进行简单捕获，需要分清稳定代码和非稳定代码，稳定代码指无论如何都不会出错的代码，非稳定代码捕获时尽量区分异常类型，再针对性的异常处理。同时不要直接用异常处理来做流程控制（包括性能损耗和代码清晰度）。模块内可以通过抛异常的方式返回数据，但模块间需要通过返回错误码的方式返回。

异常处需要采集排查问题的关键日志，并需要上报监控。

1.1.4 幂等

◆ 幂等

幂等：对同一操作发起的一次请求和多次请求的副作用一致。

分布式系统的调用失败有一种情况是等待返回时超时，对于这种情况，调用方并不知道之前的请求是否成功执行，被调用方可能执行成功也可能没执行。这时候如果接口是幂等的，调用方就可以继续发起重试请求，来确保请求能最终执行成功。

◆ 不幂等影响

Ø 业务相关

如果接口非幂等，业务该怎么办？正常情况下不会有问题。而一旦发生超时，

是没有什么好的办法。首先在没有收到明确回复的情况下是不能重试的，因为重试会导致重复执行的错误数据。其次不重试的话，态的伤害，不同的业务也各不相同。展示类的业务可能刷新下就能恢复。而相反，对于交易类的业务，尤其是多步流程中的一环，一般都需要自动对账加修复，多个环节的事务性修复可不是什么容易的又可能出现不一致状态，例如调用者按照失败来变更相关状态，但实际上数据层已成功。

Ø 容灾相关

不幂等要求服务下线是优雅的。也就是必须处理完所有请求才能关停服务。这也意味这服务突然的中断——网络故障、电力故障、机器故障等是有损的。同理，运维如果要做流量切换，也不能暴力调度。

◆场景分析

增：如果是数据库自增 ID 明显就不是幂等的，而如果是基于 ID 或者 KEY 的参数来调用的话做存在判断能做到幂等。

删：如果是基于 ID 或者 KEY 来做天然是幂等的，而如果是删除最后十条这种就天然不是幂等。

改: 直接覆盖是幂等, 增量累加不是幂等, 对应 HTTP 中 PUT 是幂等, POST 不幂等。

查：天然是幂等的。可以说只读的接口都是幂等的。

◆ **接口幂等改造方法**

Ø 唯一操作凭据；

Ø 防重放，保存一段时间非幂等操作；

1.2 架构容错，自动切（杨权新）

如果高可用只能做一件事，那一定是自动切。

上图简单描述了用户发起的请求，所经过的调用层次。

每个层次的调用都可能因为网络、机器、程序等故障而发生失败，此时就需要自动切换到容灾的备点进行调用，以保证可用性。

每个层次实现自动切都有自己的一套机制，比如 APP 层的 ACK 系统，CDN 的 Nginx backup，公共接入的 LVS，服务层的 RPC 和中间件。

1.2.1 接入层去单点与自动切

1. 客户端 ack 自动切：

Ack 容灾策略提供：多域名、域名优先解析和容灾网关多种重试机制，多域名配置调用逻辑是主域名失败重试备用域名再失败重试容灾网关；域名优先解析可以解决 DNS 域名劫持，调用逻辑是（广州用户）广州失败重试北京再失败重试容灾网关，北京用户调用逻辑反之。

2. 公共接入去单点：

公共接入由 LVS+Nginx 组成，负载均衡策略为轮训机制，业务部署多个接口机器，添加至公　共接入 Nginx Upstream 模块实现业务去单点

3. 公共接入自动切：

公共接入配置重试次数（默认 3）和故障节点自动剔除机制（http 标准错误返回码：500、502、503 和 504）实现业务自动切；如跨机房业务可配置异地机房为备点实现业务自动切

1.2.2 业务层去单点与自动切

1. 业务去单点

业务单区域单机房至少部署 2 节点，添加至公共接入 Nginx Upstream 模块实现业务去单点

2. 业务自动切

业务节点故障，返回 5xx 公共接入会利用重试策略或者故障节点自动剔除机制实现自动切，业务访问缓存或者存储失败必须返回 http 标准错误码（500、502、503 或 504）由公共接入或者客户端 ack 完成自动切

3. 业务依赖自动切

必须接入 RPC，配置超时（业务超时>RPC 超时>被调方超时）合理实现自动切。

1.2.3 数据层去单点与自动切

1．数据层去单点

非关系型数据库，利用分布式（riak）多副本策略实现去单点；关系型数据库可以利用主从或者主主架构实现去单点；缓存可以利用主从或者集群实现去单点

2．数据层自动切

关系型数据库必须接入架构组 MySQL 中间件实现自动切，分布式数据库（RIAK 等）提供故障自动转移实现自动切，缓存主从模式可以利用哨兵实现自动切，缓存集群主节点故障内部实现主从自动切

1.3 异常发现

异常发现是服务稳定的有效保证，当服务出现异常时，异常发现系统可有效地将异常信息告警给相关人员，以便及时处理，恢复服务正常。酷狗的异常发现主要有 APM 和监控平台，两者的具体用途如下:

	异常发现平台	**关键指标**
APP 层	APM	用户可直接感知的时延、失败率等
接入层	网络监控、测试监控	ACK 接口、容灾网关的连通性
接口/服务层（含 RPC)	监控平台	QPS、时延、失败率
存储层(Mysql、Redis)	监控平台	QPS、时延、失败率

1.3.1 APM 客户端监控（李毅）

APM（Application Performance Management & Monitoring）即酷狗各终端应用性能监控管理系统，它可对酷狗核心应用的关键性能指标提供实时的监控和预警，是客户端和后端服务的眼睛；当服务或客户端出现故障时，可实现分钟级的预警，给服务的稳定性提供有力的保障；可以帮研发人员分析和定位故障，为故障恢复和系统优化提供可靠的数据。

1. **APM 系统入口及权限介绍**

AMP 系统访问入口：http://apm.kugou.net/index（OPD 首页有入口）

APM 系统登录账号密码：OPD 账号及密码

APM 权限的申请流程：OPD 首页—>Eflow 流程系统—>数据服务中心—>APM 系统权限申请流程—>开通权限（根据需要填写开通范围）

2. **APM 系统基本功能模块介绍**

APM 覆盖终端：PC 端、MAC 端、移动端（Android、IOS）、繁星 WEB、独立繁星、繁星伴奏、酷狗唱唱、看、唱、KTV、后台、酷狗官网、硬件、短视频等。

APM 监控指标类型：速率、时延、可用性、卡顿率、崩溃率等。

APM 分析查询时间维度：分钟、小时、天。

APM 可分析查询指标维度：版本、运营商、地域、网络类型、综合分析等。

3. **AMP 系统使用说明**

APM 平台首页有功能介绍及详细的使用说明文档可查阅。APM 管理后台采用可视化 WEB 界面操作，简单明了，基本上是看到就会用，使用时只需根据界面菜单指引，选择对应的指标和维度，点查询即可。

4. **APM 对后端研发人员的作用(应用场景)**

1）服务更新时，查看相关指标是否有异常波动；

2）平时服务指标达到预警值告警时，通过 APM 后台分析定位原因并跟踪修复；

3）随时掌握自己负责服务的性能数据，为服务优化提供依据。

5. **APM 高级应用**

通过 APM 管理后台定位问题时，有时不能获得完整的信息，我们可以通过查询原始上报数据来获得更详细的信息。

APM 原始上报数据查询入口：

http://10.12.0.22:11116/beeswax/execute/query/1106#query/results

公共帐号：public

密码：public123

Sql 语句的获取：在选择好指标、维度、时间以后，点查询，然后点击复制 sql，即可得到当前查询的 sql 语句。粘贴至 hue 上查询即可。若需查询其他内容，不了解如何写 sql 的，可咨询周徐波或者黄日梅。

1.3.2 接口监控（李毅）

接口监控平台是云技术部接口服务稳定性和性能数据采集、展示、告警的统一平台，它肩负着服务性能的实时监控重任，是云技术部一个重要的基础公共服务。

监控平台可以采集和展示接口的响应时长、QPS、5XX、响应内容大小、http 状态码、APPID、SERVERID、错误码、版本号等通用信息，也可展示业务上报

的自定义监控指标，如：注册量、劫持量、各种耗时、RPC 的 QPS 等；还可展示 RPC 和探针相关数据。

云技术部后端接口服务要全部接入监控平台上报，通用数据上报和采集直接在 nginx 上通过采集脚本收集上报，业务无需关注，自定义指标上报、RPC、探针数据上报需要业务方根据上报接口埋点。

自定义上报对接人：黄志坚

RPC、探针数据上报对接人：吴德伟

接口监控平台地址：

http://kgmonitor.kugou.net/monitorAdmin/index/index/

接口监控平台使用说明文档：

http://172.17.10.161/TestStatistics/dataaggregation/use.pdf

接口监控平台登录账号密码：OPD 账号及密码

接口监控平台开发维护负责人：黄志坚、宋坤

1.3.3 调用监控（杨权新）

1. **RPC 监控**

监控信息包括有：调用哪些外部服务接口，以及故障业务是否成功自动切。

入口：opd 首页->业务接口监控平台->具体业务（个性化推荐->猜你喜欢电台）->探针报表->看到 rpc 字样->查看详细。

备注：rpc 详细内容可以查看 http://git.kugou.net/kugou-rpc/document

2. **Redis 监控**

查看 redis 监控指标包括:kmc 监控平台和 redis 监控平台，kmc 监控平台不支持集群维度，入口：opd 首页->KMC 监控系统->具体业务（个性化推荐->猜你喜欢电台）->DB->Redis->Redis 状态->选择查看具体实例 ip&端口；

Redis 监控平台可以查看 redis 集群状态，功能包括：告警配置，业务可以自行修改告警指标和短信收敛频率，入口：http://s.redis-monitor.kugou.net/

3. **Mysql 监控**

查看 mysql 监控数据入口：opd 首页->KMC 监控系统->具体业务（个性化推荐->猜你喜欢电台 DB）->DB->Mysql。

4. **Mysql 中间件**

查看 mysql 中间件监控指标入口：opd 首页->业务接口监控平台->具体业务（数据库云化->账号风险->详细信息）。

5. **MQ 消息中间件**

查看 MQ 消息堆积等监控指标入口：opd 首页->KMC 监控系统->云技术部->基础公共服务->消息队列中间件->指标->指标列表->具体指标（详细信息区分队列）。

6. **灰度系统监控**

查看已接入架构组灰度系统监控指标入口：opd 首页->业务接口监控平台

->

7. **Riak 监控**

查看 Riak 监控指标入口：opd 首页->KMC 监控系统->云技术部->具体业务（用户账号->个人中心 riak 集群->DB->Riak)。

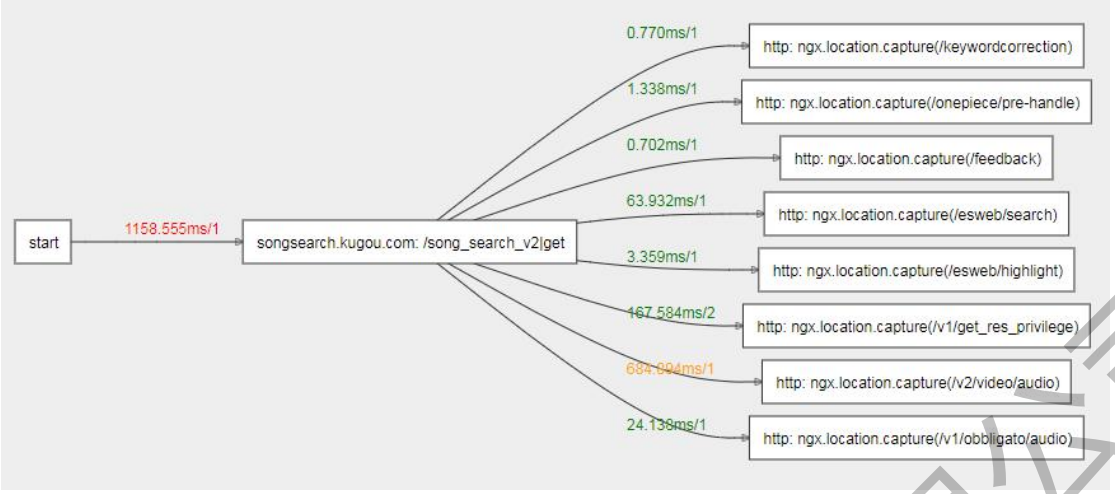
8. **探针监控**

查看业务调用 redis、myql、riak 和依赖接口性能指标监控入口：opd 首页->业务接口监控平台->具体业务（个性化推荐->猜你喜欢电台->业务后端机->探针报表)。

1.3.4 调用链分析（戴育东）

1. 调用链主要作用：发现调用过长的链路，发现耗时过长的调用
2. 调用链监控平台：http://callgraph.kugou.net
3. 调用链接入：业务运维
4. 调用链功能：链路状态监控，时延监控，涵盖 HTTP、redis、mysql、memcached

举例：通过域名"详细列表"发现搜索服务一个超过 1s 的请求，再通过"依赖图"可以发现调用"/v2/video/audio"耗时较高



1.3.5 异常大屏幕统计（李毅）

异常大屏酷狗后端服务异常信息的集中展示，是监控平台对异常监控的有效补充，异常大屏会按照最近 24 小时的异常上报量展示对应的异常指标，可根据设置的预警值对超出预期的异常进行预警。

云技术部所有核心接口服务都要接入异常上报，异常上报采用接口服务自己写异常日志，采集脚本自动解析日志并上报的策略，采集脚本把异常上报内容分类合并后同步到监控平台进行异常信息展示和告警，同时将数据上报给 BI 展示到异常大屏。

需要上报到异常大屏的场景举例：客户端请求参数错误、入参 XXX 类型不合法、XXX 无权限、调用 XXX 超时、数据格式错误、最后一个 else 分支异常信息捕获、default 分支异常信息捕获、接口返回的数据量与预期不符（例如：top100 返回数据数量少于 100 个、电台/专辑返回内容为空、BI 获取到的数据为空等） ... 总之，程序员觉得有必要的异常都要上报。

埋点上报流程：

1）开发人员去错误码管理后台申请错误码

异常/错误码【exceptionid】申请管理后台（负责人：刘明桃）：

http://kgedit2.kugou.com/kugouAdmin/src/

菜单位置：异常管理中心>>错误码管理后台

字段简单说明：

>>平台：云技术部后端服务

>>所属团队：业务所属开发组（例如：后端开发 1 组、后端开发 2 组、基础架构组，等）

>>所属模块：业务模块中文名，与 serverid 对应

>>操作人：添加数据的人员，请写自己

>>影响类型：下拉列表选择合适的类型

>>错误信息：错误描述，文字要少，要外行（不了解细节的人）能看懂

2）按照“服务端上报接口及规范”在服务端代码中埋点写异常日志，走测试流程

服务端上报接口及规范：http://basewiki.kugou.net/doku.php?id=工作规范:服务端异常日志上报规范

3）服务端代码上线后通知运维部署异常上报脚本

异常上报脚本接口人：李永昌、黄志坚

4）测试人员确认异常上报

1.3.6 ELK 业务日志查询平台（任思豪）

日志查询平台的好处：

统一日志，免去需登录多台机器查看日志的麻烦。

快速查找符合指定条件的日志，如查找某用户的请求日志。

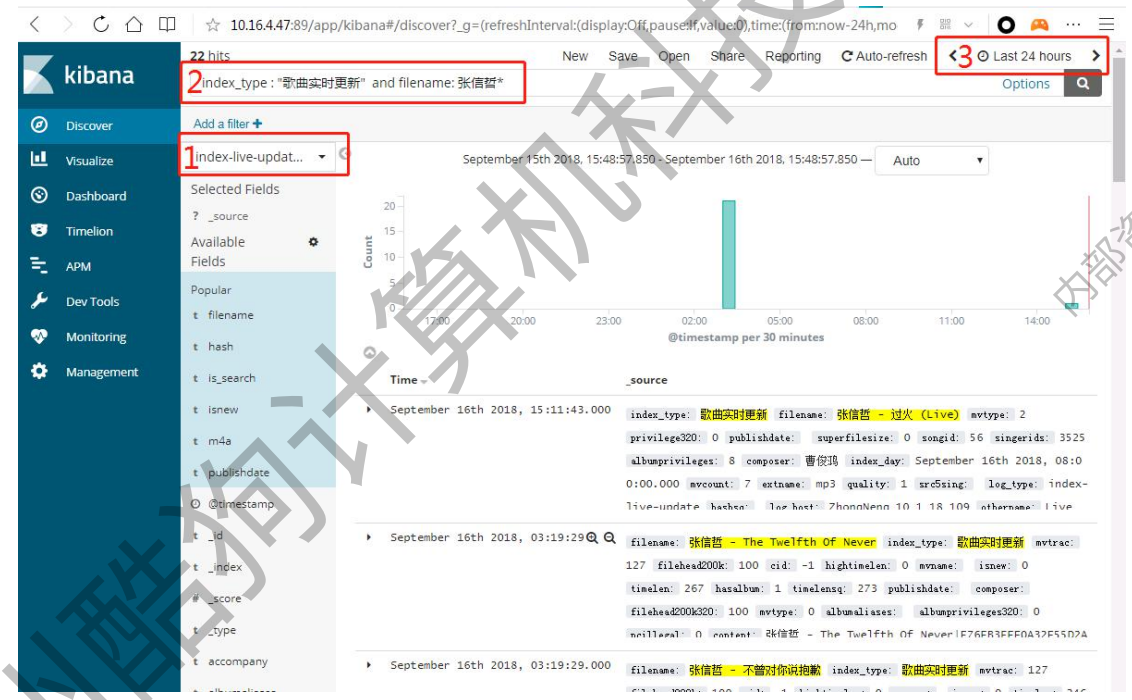
酷狗 ELK 业务日志查询平台地址：

http://10.16.4.47:89/app/kibana#/discover?_g=()

常用功能：

1）查询指定条件日志示例

eg：查询最近 24 小时内，更新的关于张信哲的歌曲。



1. 在上图 1 的地方选择要查询的业务日志索引，本例子选择 index-live-update-*。

2.在上图 3 的地方选择要查询的时间段，本例子选择 last 24 hours。

3.在上图 2 的地方输入查询的条件，支持用 and/or 来多条件组合查询， 本例

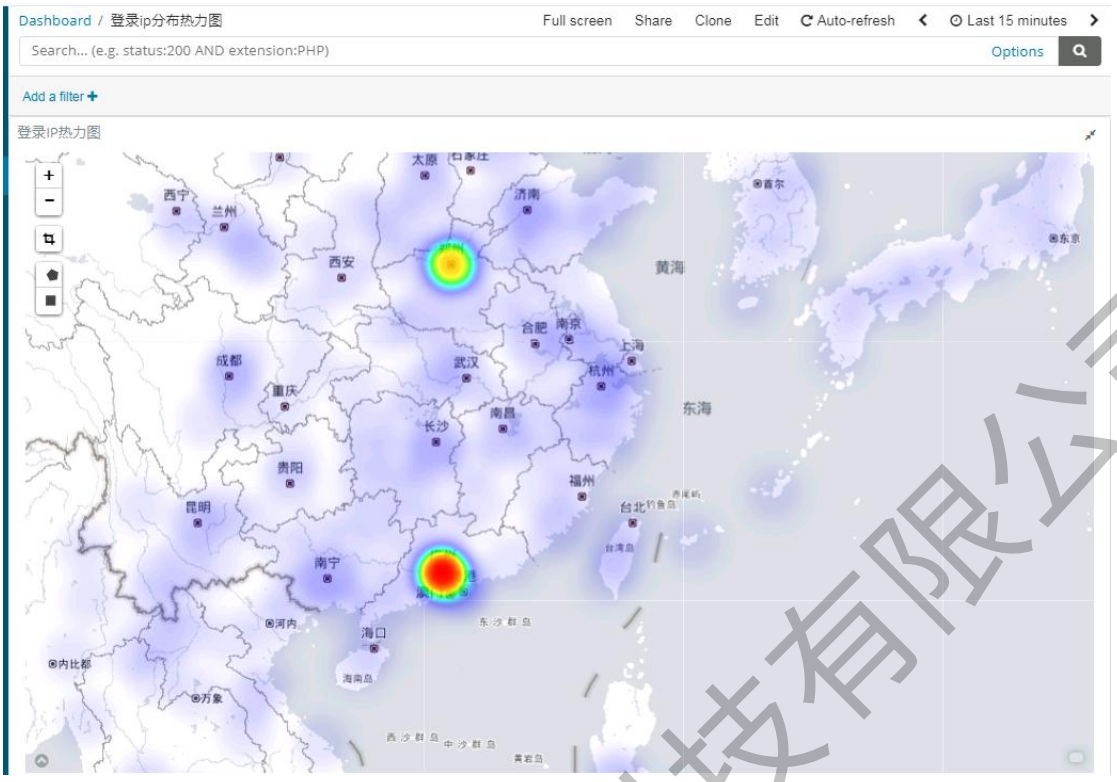
子是查张信哲的歌曲变更。所以输入的是：index_type : "歌曲实时更新" and filename: 张信哲*

2）查看日常监控的仪表

目前 ELK 支持多种图表的仪表，有常用的柱形图、折线图、饼图等，也支持词云，地图热力分布图等。

1. 近 15 分钟单曲搜索的 topN 关键字

2. 近 15 分钟登录的 IP 分布



1.3.7 BI 流水分析（任思豪）

数据查询后台 HUE 地址：

http://10.16.4.179:8888/home#

客户端 BI 流水表名分别是：

PC： dt_list_pc_d

安卓：dt_list_ard_d

iOS：dt_list_ios_d

bi 数据查询，支持类 sql 语句。数据以日期分区，所以查询条件需加上 dt='yyyy-mm-dd'。

eg： 查询 PC 2018-05-01 到目前，每天的搜索播放量

select dt,count(*) as total from ddl.dt_list_pc_d where dt >= '2018-05-01' and action='play' and fs<>'播放错误' and cast(spt as int) >0 and length(user_id)<>0 and length(mid) in (32,36,40) and (fo_2 like '搜索结果%' or fo_2 like '综合搜索%') and sty='音频' group by dt limit 100

其他常用字段：i 为用户 id，tv 为客户端版本号。

KPI 流水、行为流水、无埋点流水

1.4 故障处理（戴育东）

1.4.1 快速定位问题

1. APM 监控：http://apm.kugou.net/index

   需要依次分析：

   1) 可用性，分析失败率同比

   2) 错误原因，分析错误码与失败用户，是否具备个人用户特征?

   3) 版本分析，是否 app 新发版导致?

   4) 地域分析，是否某个地区用户受损?

   5) 运营商分析，是否某个运营商用户受损?

   6) 安卓端和 iOS 端是否同时受损?

2. 平台监控：http://kgmonitor.kugou.net/monitorAdmin/chart/index

   需要依次分析：

   1) 5xx 请求是否异常

1）响应时间是否异常

1）QPS 是否异常

1）错误码是否异常

1）rpc 调用是否异常（失败率/时延）

1）探针是否发现调用异常（mysql/redis 等）

3. KMC 监控：http://opd.kugou.net/kmc/

需要依次分析：

1） 机器指标（CPU、内存、磁盘、网卡、socket）

2） redis 监控/mysql 监控

3） 机房网络监控

4. 平台 Redis 监控：http://s.redis-monitor.kugou.net/

1）节点状态

1）慢查询

1）内存使用率

1）qps

2. QA 反馈：http://qa.kugou.net/

3. 调用链监控：http://callgraph.kugou.net

小结：

1. 通过各监控渠道快速发现问题，主要依赖 **APM/监控平台/KMC**

2. 询问开发是否有执行变更，包含但不限于（代码发布/配置发布/上机操作）等

3. 询问运维是否有执行变更，包含但不限于（域名切换/RPC 切换/证书切换/上机操作）等

1.4.2 快速恢复

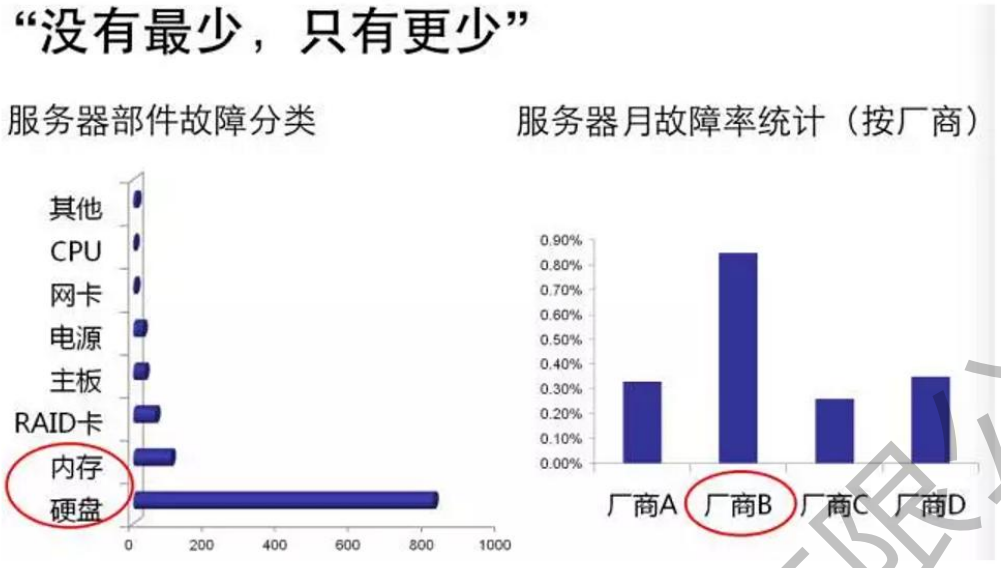
1. 若是发布相关引起故障，即时回滚
2. 若是地区网络故障/机房网络故障/机器故障，且 15min 内无法恢复，切流/摘点
3. 若是被刷或者被恶意攻击，通过调整天幕阀值自动降级，或屏蔽来源 IP
4. 若引起雪崩，向部门负责人申请后，可暂时屏蔽入口

1.4.3 复盘，故障报告

1. 故障恢复后，应 48 小时内完成复盘，以免线索丢失；
2. 根据规范（http://basewiki.kugou.net/doku.php），拟写故障报告，回复监控日报或 QA 日报；
3. 另将故障报告发送至部门相关负责人，用于故障月会回顾以及存档。

1.4.4 常见故障分类与处理（待补充）

常见的故障有：硬件故障、软件 BUG、网络波动或中断、容量性能不足等等。

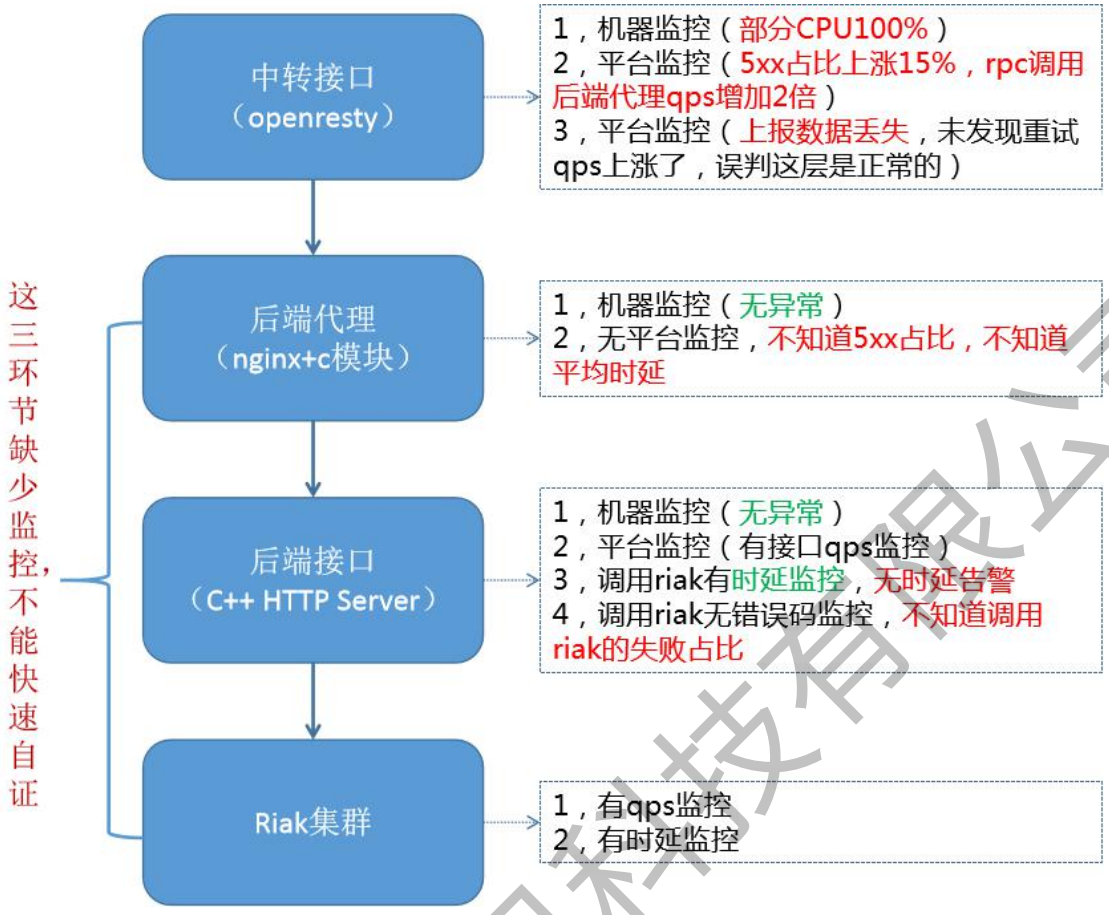


1.4.5 故障举例

【分析】

2018 年 11 月 02 日，网络收藏同步和上传失败率上涨，影响时长约 4 小时，约影响 700 万用户（大部分用户感知）

网络收藏架构简图（故障时）：



【故障经过】

18:41，广州 Riak 集群波动，中转接口时延加大，失败率开始上涨，**APM 触发告警**；

18:44，广州中转接口机器 CPU 使用率上涨至 99%，北京中转接口机器 CPU 使用率上涨至 90%；

由于中转接口上报数据部分丢失，开发误判中转接口是正常的。后端代理/后端接口/Riak 的 5xx/错误码监控告警不完善。由于缺少各层级 5xx 数据对比，开发无法快速定位这三个模块有没有问题，花了 2 个小时排查与扩容这两个模块，这些措施都无效。

21:26，扩容北京 3 台中转接口机器失败；

21:58，杀掉一台中转接口机器的 Openresty 僵死进程，CPU 使用率略有下

降；

22:23-22:25，杀掉所有中转接口机器的 Openresty 僵死进程；

22:28，服务恢复。

【总结】

问题一：openresty 低版本存在 Bug，单个进程内存使用量超 2G 后，会进入死循环。

问题二：由于**缺少监控**，开发不能进行各个层级的 5xx 数量对比，时延对比，**定位问题慢。**

问题三：中转接口单 CPU 长时间用满，但**未发出告警**，未能及时查出 openresty 进程僵死。

问题四：扩容新机自动化不足，中转接口扩容慢，也导致了故障恢复慢

1.5 变更的规范与流程

1，必须走 OPD 流程

线上的变更，包括变更程序、代码脚本、配置，必须走 OPD 流程。审批至“部署”阶段才能开始操作。

对线上环境有敬畏之心，严禁不走流程直接修改代码脚本。

OPD 的测试审批环节，原则上不允许开发自己审批，除非得到上级授权。

2，禁止搭便车行为（变更需求无关代码）

可以提多个需求，合并变更。但禁止搭便车的“优化”。

3，核心服务变更，必须提供一键回滚

在设计和施工阶段都应该保证方案实施过程的可回滚。为保可用性，这点不允许商量。

3.1，关系型数据库，禁止修改线上库的表字段，只允许增加字段，或者增加表。

3.2，已接入 RMS 发布系统的，禁止手工发布。

4，核心服务，禁止直接全量，必须先灰度至少半天

除非业务原因，得到上级特批。

5，遵循观测指标，回滚标准，上报标准

变更过程发现指标异常，请按标准及时回滚和上报。

标准请遵守文件:《核心服务观测指标&回滚上报标准 - 云技术部 - 2017.xlsx》

违反惩罚

违反规范，但未造成不良后果的，季度考核扣 5 分。

违反规范并造成不良后果的（包括用户体验及公司资产声誉损失等等），季度考核扣 10 分。

违反规范并造成严重不良后果的，季度考核直接为 C。

1.5.1 设计评审

一人计短，众人计长。

在重大需求、服务重构、组件设计的方案把握上，仅仅依赖一个开发是风险很高的事情。比如红包广场需求，支付牌照的风险就一直到最后才由评审发现。但已经浪费了三个月的开发人力投入。

因此，在超过两周的开发人日投入的项目，均应举行设计评审。开发 leader、架构师、运维、测试为必参人员，并具有方案否决权。

1.5.2 Codereview ？求补充

1.5.3 测试（李毅）

为了保证交付质量，所有开发任务发布前均要通过测试环节，这里的测试不是单指测试人员的测试，也包括需求方自测和开发自测。

1．常规测试流程

代码 Review -> 单元测试(开发) -> 开发自测(功能、性能) -> 需求方验收测试 -> 测试用例评审(开发、产品、测试同行、性能测试方案需要运维参与评审) -> 测试环境测试 -> 预发布环境测试 -> 线上验收测试。

2．产品、开发自测流程

简单的技术优化需求、内部管理后台需求、导数据类需求、修改文案、修改图片类需求因风险较小，经开发人员或测试人员评估，认为可由需求方或开发自测的，自测通过后即可直接上线，负责自测的开发或需求方必须执行测试，严禁不经测试直接上线。

1.5.4 灰度（杨权新）

1. **核心业务灰度策略**

单机房灰度 1 台机器，观察指标如异常立马回滚，如正常灰度 1 天，逐渐加量，单边全量至少观察 1 周。

2. **非核心业务灰度策略**

单机房灰度 1 台，观察指标如异常立马回滚，如正常灰度 1 天，单边全量至少观察 2 天。

1.5.5 小步迭代（梁海龙、李毅）

小步迭代，指的是遇到较大需求或改版时，不要直接长时间研发并直接发布超大版本。因为改动越大，积累的 BUG 和不确定风险就越大。为了高可用，需要把大需求拆分为可演进的小需求。同时按小的演进步骤逐步发布小阶段式变更，迭代这个逐步发布的过程来完成整个大需求。逐步演进可以在过程中逐步验证并消除风险，而避免了累积风险并爆发的后果。（PS：这个理念和单元测试类似，单元测试也能逐单元验证并避免累积风险）

另外为了确保研发效率和交付周期，持续交付，要求尽量将任务细化拆分，能独立的任务尽量独立开来，建议按照 1 人日的规模来拆分任务，严禁需要超过 3 人日的任务。我们要求减少任务间的耦合度，确保每个任务都是可以独立发布到线上进行验证的个体，这样可减少集中发布的风险，提升研发效率，缩短研发周期和交付周期。

1.5.6 标准操作流程（李毅）

必读：《核心服务观测指标&回滚上报标准 - 云技术部 - 2017.xlsx》

1. MTP 流程：所有发布的需求都需要在 MTP 上有记录可查。

2. OPD 审批流程：OPD 须满足以下要求才可通过测试审批。

1）必须明确要上线的任务地址（MTP/BUG）；

2）MTP 对应的任务必须是“线上部署”状态；

3）灰度上线和全量上线 OPD 需分开提；

4）OPD 必须包含审批人自己负责的测试任务（其他人负责测试的，批单人需要找对应的测试确认，并拉相关人员进上线讨论组一起沟通）。

要求：不满足要求的 OPD 测试一律拒绝。

3. 测试线上验证流程：

1）原则上必须当天完成线上验证，完成 OPD 和 MTP 状态更新；

2）若开发未及时处理 OPD 或 MTP 状态，测试必须及时提醒直至按要求处理；

3）灰度上线（OPD 必须体现）完成，验证通过，OPD 和 MTP 状态就可更新，下次灰度时需要重新提 OPD；

4）因紧急发布或特殊申请的，允许先完成任务后补 MTP 和 OPD 流程，时限为任务完成后的第一个工作日内。

1.5.6 回滚（转自微信文章）

高可用服务在变更时必须提供回滚！ 这点不允许商量！ 这么重要的话题，我还要用一个感叹号来强调一下！

但是估计现实情况里，大家都听过各种各样的不能回滚的理由吧。我这里有三条买也买不到的秘籍，今天跟大家分享一下，保证药到病除。

理由 1：我这个数据改动之后格式跟以前的不兼容了，回退也不能正常！

秘籍 1：设计、开发时候就考虑好兼容性问题！！！比如说数据库改字段的事就不要做，改成另加一个字段就好。数据存储格式就最好采用 protobuf 这种支持数据版本、支持前后兼容性的方案。最差的情况，也要在变更实施『之前』，想清楚数据兼容性的问题。没有回滚脚本，不给更新，起码做到有备而战。

理由 2：我这个变更删掉东西了！回退之后数据也没了！

秘籍 2：你一定是在逗我。把这个变更打回去，分成两半。第一半禁止访问这个数据。等到发布之后真没问题了，再来发布第二半，第二半真正删掉数据。这样第一半实施之后需要回滚还可以再回去。

理由 3：我这个变更发布了之后，其他依赖这个系统的人都拿到了错误的数据，再回退也没用了，他们不会再接受老数据了！

秘籍 3：这种比较常见出现在配置管理、缓存等系统中。对这类问题，最重要的就是， 应该开发一种跟版本无关的刷新机制。触发刷新的机制应该独立于发布过程。 要有一个强制刷新数据的手段。

以上三个秘籍覆盖了 100%的回滚兼容性问题，如果有不存在的，请务必告诉我！

回滚兼容性问题，是一个整体难题。只有开发和运维都意识到这个问题的严重性，才能从整体上解决这个问题。而解决不了回滚难题，就不可能达到高可用。

2 高性能

2.0 综述（梁海龙）

性能取决于**硬件**和**软件**。

软件性能的本质，在于**算法数据结构**与**并发控制**。

优秀的程序员不仅对自己的设计编码有性能认识，同时对依赖的各类系统调用、第三方库也有性能数量级的了解。

并发的性能要素之一，是资源访问的**竞争处理**。大锁还是小锁？有锁还是无锁？分布式下如何处理竞争来保证数据一致性？

并发的性能要素之二，就是**并发模型**——任务是以什么样的形式执行，以及这些执行如何触发调度。比如进程、线程、协程都是可执行体。比如任务消费线程池、事件驱动模型、Reactor、Proactor 等等都是触发调度的不同模型。

性能优化最重要的是找到**瓶颈**，所有在瓶颈以外的优化都是耍流氓（无效优化）。

性能优化的收益请参考 Amdahl Law（阿姆达尔法则，如下公式），p 是局部模块的使用率占比——比如延迟占比，小 s 是局部模块性能优化的倍数，大 S 是总体性能优化的收益：

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

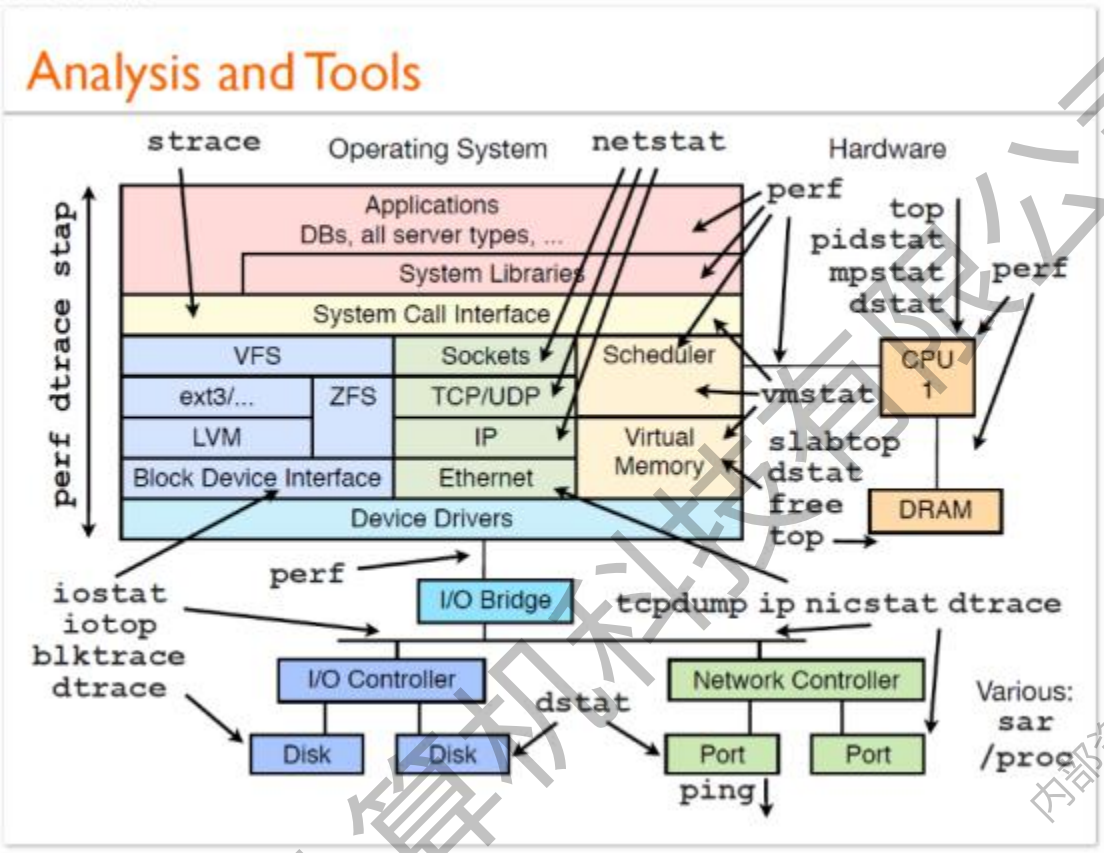
性能管理最重要的是**度量**。推荐《性能之巅》，很好的阐述了 linux 系统的各层性能要素和度量工具。但最重要的还是生产流程和生产环境建立**性能度量和监控体系**。比如在上线前的压测以确定系统容量。比如在大促之前，做线上流量局部压测或全链路压测。比如在系统发生问题之时，能通过监控系统很快的定位到问题节点和问题组件。

表 2.2　系统的各种延时

事件	延时	相对时间比例
1 个 CPU 周期	0.3 ns	1 s
L1 缓存访问	0.9 ns	3 s
L2 缓存访问	2.8 ns	9 s
L3 缓存访问	12.9 ns	43 s
主存访问（从 CPU 访问 DRAM）	120 ns	6 分
固态硬盘 I/O（闪存）	50–150 μs	2–6 天
旋转磁盘 I/O	1–10 ms	1–12 月
互联网：从旧金山到纽约	40 ms	4 年
互联网：从旧金山到英国	81 ms	8 年
互联网：从旧金山到澳大利亚	183 ms	19 年
TCP 包重传	1–3 s	105–317 年
OS 虚拟化系统重启	4 s	423 年
SCSI 命令超时	30 s	3 千年
硬件虚拟化系统重启	40 s	4 千年
物理系统重启	5 m	32 千年

2.1 指令级别性能（杨权新）

2.1.1 Performance 工具图



2.1.2 火焰图

1. **优点：**

(1) 能够直观地显示进程内函数调用关系。nginx 类(nginx,tengine,openresty)的应用首选。

2. **缺点：**

(1) 在某些版本 linux(centos7.x)可能会运行失败。此时可以考虑使用 gperftools。

3. **使用方法：**

(1) opd 提单让运维在目标机器安装火焰图。

(2) 调用以下命令获得函数调用关系。

• openresty 下查看 lua 代码的调用：ngx-sample-lua-bt

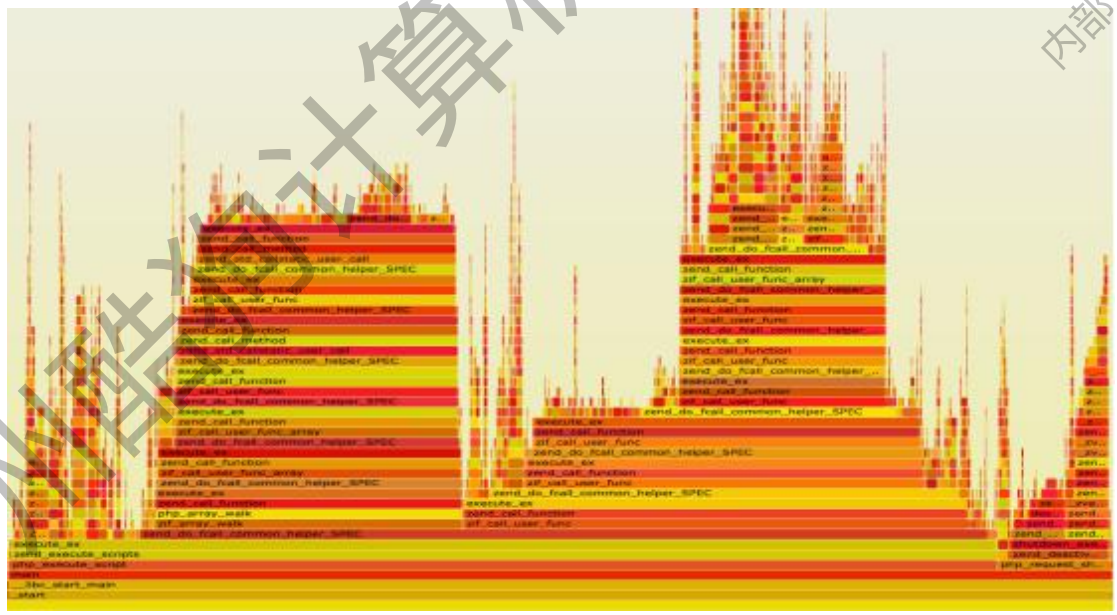
• 非 openresty 的应用查看代码的调用：sample-bt

• 查看 nginx 的代码阻塞情况（适用于第三方模块开发）：ngx-sample-bt-off-cpu

• 查看非 nginx 的代码阻塞情况: sample-bt-off-cpu

其它更多的命令介绍见：https://github.com/openresty/openresty-systemtap-toolkit。

(3) 假设步骤 2 产生的文件为 tmp.bt，依次使用链接里的工具(运维安装)：./fix-lua-bt tmp.bt>a.bt stackcollapse-stap.pl a.bt>a.cbt flamegraph.pl a.cbt > a.svg 最后生成的 svg 文件在网页打开即可。



上图中色块的“长度”表示该函数占用的 cpu 时间，色块的“叠加”表示函数间的调用关系。色块的颜色则没有什么特殊意义。

4. **Go 程序生成火焰图：**

(1) 业务在代码中加入以下语句：

```
import _ "net/http/pprof"
```

在 main 方法中增加

```
func main() {
    go func() {
        http.ListenAndServe("localhost:6060", nil)
}()
    //业务代码
}
```

(2) 然后对该应用的 6060 端口发起以下请求，即可以生成函数调用关系：

http://localhost:port/debug/pprof/，除了可以看 cpu 调用关系，还可以看内存等消耗情况，具体的使用指令请见：

https://www.cnblogs.com/ghj1976/p/5473693.html

(3) 安装 go-torch https://github.com/uber/go-torch，然后使用 go-torch 命令生成火焰图：

```
./go-torch -u http:// localhost:6060/debug/pprof/profile -t 120
```

2.1.3 字符串拼接导致的 w 级 QPS 骤降到 k 级

1．lua 字符串处理需要注意以下几点：

• lua string 是常量，所有 string 被保存在一个 vm 级别的哈希表里面，创建

string 需要先生成 hash，再去查找哈希表有没有重复 string，这是因为 lua 为了提高 string 作为 table key 的效率而设计的，但带来的后果就是创建 string 很低效，尤其是长字符串，又要 hash 又要扫描。所以减少生成字符串的次数。

• 对于数目预知的短字符串（或数字类型）拼接，直接使用拼接操作符，这样相当于手工的 unroll loop，而且不需要循环的情况下，luajit 可以预知拼接元素的类型。

• 对于循环(尤其是长字符串）拼接操作，借助 table 来生成字符串更高效。

错误例子：

```
function bar1(a,b,c)
    local s = ""
    s = s..a
    s = s..b
    s = s..c
    return s
end
```

正确例子：

```
function bar2(a,b,c)
    local t={}
    t[1],t[2],t[3] = a,b,c
    local s = table.concat(t,"")
```

```
    return s
end
```

2． openresty+lua 不适合写 cpu 密集的应用。比如：频繁把长字符串进行 json encode/decode。

• openresty 是多进程模型，其中每个进程又是单线程运行，依靠 epoll_wait 触发网络事件或者定时事件的响应。假如业务逻辑是通过网络接收一个很大的字符串(比如几 M byte)，然后对其还原为 json 格式。此过程会一直占有 cpu，比如 10ms，那么该 10ms 内内核接收的网络数据包将延后处理，“被迫”增加 10ms 时延。如果业务频繁处理这种长字符串，越往后到的数据包将“看起来时延很大”，甚至有可能超时。

• Openresty 的时间更新机制是在处理完 epoll_wait 所有网络数据包之后，如果在处理网络数据包占用的 cpu 过长，则 openresty 的定时任务 ngx.timer 和 openresty 应用层会报大量的超时，此时需要通过捉包(tcpdump)来定位是否网络回包慢。

3．openresty 更多的“坑”见附件。

"此处无法插入文件对象"

2.2 并发事务（李长豪）

2.2.1 锁，CAS

在多线程编程中，为了保证数据操作的一致性，操作系统引入了锁机制，用

于保证临界区代码的安全。通过锁机制，能够保证在多核多线程环境中，在某一个时间点上，只能有一个线程进入临界区代码，从而保证临界区中操作数据的一致性。

悲观锁：认为每次获取数据的时候数据一定会被人修改，所以它在获取数据的时候会把操作的数据给锁住，这样一来就只有它自己能够操作，其他人都堵塞在那里。

乐观锁：认为每次获取数据的时候数据不会被别人修改，所以获取数据的时候并没有锁住整个数据，但是在更新的时候它会去判断一下要更新的数据有没有被别人修改过，例如更新前查询该数据的版本号，更新的时候看看该版本号有没有被人修改过，如果被人修改过了，那就不会去更新。

CAS 语义（比较与交换，Compare and swap） 是乐观锁的核心，也是实现无锁编程的关键。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。实现非阻塞同步的方案称为“无锁编程算法”（ Non-blocking algorithm）。

CAS 指令其实是一个 CPU 的指令，它会拿内存值和一个给定的值进行比较，如果相等的话就会把内存值更新为另一个给定的值。其实 CAS 指令就是单机使用一个乐观锁的机制。

举例 MySQL 的 CAS 乐观并发锁，解决分布式数据库的读写冲突：

https://www.cnblogs.com/grefr/p/6088009.html

举例 Java 中使用 CAS 指令，实现单机无锁编程：

https://blog.csdn.net/sinat_34976604/article/details/80970939

2.2.2 事务

◆ 代码实现的事务可靠吗

代码实现的事务并不可靠，应该把数据操作封装到数据库事务中。

◆ Mysql 事务级别

◆ Read Uncommitted(读取未提交内容)

1. 所有事务都可以看到其他未提交事务的执行结果
2. 本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少
3. 该级别引发的问题是——脏读(Dirty Read)：读取到了未提交的数据

◆ Read Committed(读取提交内容)

4. 这是大多数数据库系统的默认隔离级别（但不是 MySQL 默认的）
5. 它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变
6. 这种隔离级别出现的问题是——不可重复读(Nonrepeatable Read)：不可重复读意味着我们在同一个事务中执行完全相同的 select 语句时可能看到不一样的结果。

导致这种情况的原因可能有：

7. 有一个交叉的事务有新的 commit，导致了数据的改变;
8. 一个数据库被多个实例操作时,同一事务的其他实例在该实例处理其间可能会有新的 commit。

◆ Repeatable Read(可重读)

9.这是 **MySQL 的默认**事务隔离级别

10. 它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行

11. 此级别可能出现的问题——幻读(Phantom Read)：当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行

12. InnoDB 和 Falcon 存储引擎通过多版本并发控制(MVCC，Multiversion Concurrency Control)机制解决了该问题

◆ Serializable(可串行化)

13. 这是最高的隔离级别

14. 它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。

15. 在这个级别，可能导致大量的超时现象和锁竞争

◆ 分布式事务

分布式事务是指会涉及到操作多个数据库的事务。其实就是将对同一库事务的概念扩大到了对多个库的事务。目的是为了保证分布式系统中的数据一致性。分布式事务处理的关键是必须有一种方法可以知道事务在任何地方所做的所有动作，提交或回滚事务的决定必须产生统一的结果（全部提交或全部回滚）。

在分布式系统中，各个节点之间在物理上相互独立，通过网络进行沟通和协调。由于存在事务机制，可以保证每个独立节点上的数据操作可以满足 ACID。

但是，相互独立的节点之间无法准确的知道其他节点中的事务执行情况。所以从理论上讲，两台机器理论上无法达到一致的状态。如果想让分布式部署的多台机器中的数据保持一致性，那么就要保证在所有节点的数据写操作，要不全部都执行，要么全部的都不执行。

一台机器在执行本地事务的时候无法知道其他机器中的本地事务的执行结果。所以他也就不知道本次事务到底应该 commit 还是 roolback。所以，常规的解决办法就是引入一个“协调者”的组件来统一调度所有分布式节点的执行。

在互联网公司应用较多的是 TCC 分布式事务模型。TCC 其实就是采用的补偿机制，其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段：

16. Try 阶段主要是对业务系统做检测及资源预留

17. Confirm 阶段主要是对业务系统做确认提交，Try 阶段执行成功并开始执行 Confirm 阶段时，默认 Confirm 阶段是不会出错的。即：只要 Try 成功，

Confirm 一定成功。

18. Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。

2.2.3 柔性事务（梁海龙）

为了解耦，互联网系统往往使用柔性事务来解决业务问题。

例如典型的会员交易事务，支付和会员开通往往不在同一系统，仅用柔性事务保证最终一致性（给了钱一段时间后一定会开通会员）和可回滚性（会员开通失败可退钱）。

通过异步的定时线程来对 MySQL 进行扫描（或订阅 Binlog），进行事务完成确认（确保已支付的订单最终开通了会员），是一种较为通用的处理方案，可归类为**异步确保型**柔性事务。

这种方案同样可以解决 MySQL 数据库和 Redis 缓存之间的数据最终一致性问题。比如可以利用 Canal 订阅 MySQL 的 Binlog，异步更新 Redis。消息队列的确认机制可以保证事务一定得到执行。这个问题如果用写代码的方式来做，比如先修改数据库再删除缓存，其实是可能中断的——比如暂时网络波动删除缓存失败。

此处的更新也要注意是否有并发覆盖问题。方案 A，用分布式排他锁来保证无并发更新。方案 B，更新订阅队列里面不保存值，仅仅取 Key 作为需要更新的通知。更新任务执行者负责读取在线值进行更新。方案 B 能保证最终一致性，但任何时刻数据都可能因为顺序问题而不一致。方案 C，更新队列或方案 B 的基础上使用

时间戳来解决冲突，即更新时使用 CAS 语义丢弃过期值。方案 C 相对优雅，并且不存储旧覆盖新问题。谷歌就采用了原子钟来实现方案 C，作为其分布式数据库的基石。

2.3 线程模型与 IO 多路复用（戴育东）

引言：高性能是程序员的技术追求，而高性能又是最复杂的一环，操作系统、CPU、内存、磁盘、网络、编程语言、架构，每个都可能影响系统的性能。高性能的关键之一就是服务器采取的网络编程模型，涉及到两个设计点：

1）服务器如何管理连接

2）服务器如何处理请求

以上两个设计点最终都和操作系统的 I/O 模型及进程模型相关

2.3.1 单线程/多线程/协程/单进程/多进程

19. 单线程：并无并发锁，方便编程，应避免单个处理大耗时操作，例如 redis 应避免大 value 操作。
20. 多线程：需要了解互斥锁/读写锁等线程同步机制的应用场景，避免不加锁造成的数据一致性问题，同时对锁的应用也要避免出现死锁。
21. 多进程：需要了解进程优先级(nice 值)，理解 copy-on-write、孤儿进程和僵死进程。

注：单线程/多线程/多进程，可以看《unix 环境高级编程》系统性学习。

22. 协程：本质是用户自己控制一个任务遇到 IO 阻塞了就切换另外一个任务去执行，与操作系统无关。常见有 lua+coroutine/python+gevent/go+goroutine，应避免使用阻塞函数。协程最大的好处是将异步+回调方式的编码对程序员不可见，程序员只需要实现更容易理解的同步方式，提高了编码效率。但这里要注意的是，使用协程时要避免出现阻塞式的函数调用。
23. openresty 中尽量避免的阻塞调用
    a. 高 CPU 的调用（压缩、解压缩、加解密）
    b. 高磁盘操作（文件读写）
    c. 非 openresty 提供的网络操作（luasocket 等）
    d. 系统命令调用（os.execute 等）

2.3.2 同步/异步

同步和异步关注的是**消息通信机制**

24. 同步：就是在发出一个"调用"时，在没有得到结果之前，该"调用"就不返回。但是一旦调用返回，就得到返回值了。换句话说，就是由"调用者"主动等待这个"调用"的结果。
25. 异步：则是相反，"调用"在发出之后，这个"调用"就直接返回了，所以没有返回处理结果。换句话说，当一个异步过程调用发出后，"调用者"不会立刻得到处理结果。

举例：

同步：你打电话问书店老板有没有《分布式系统》这本书，如果是

同步通信机制，书店老板会说”你稍等，我查一下"，然后开始查啊查，等查好了（可能是 5 秒，也可能是一天）告诉你结果（返回结果）。

异步：而异步通信机制，书店老板直接告诉你“我查一下啊，查好了打电话给你”，然后直接挂电话了（不返回结果）。然后查好了，他会主动打电话给你。在这里老板通过“回电”这种方式来回调。

2.3.3 阻塞/非阻塞

阻塞和非阻塞关注的是**程序在等待调用结果**（消息，返回值）时的状态

26. 阻塞：阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。
27. 非阻塞：非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

举例：你打电话问书店老板有没有《分布式系统》这本书，你如果是阻塞式调用，你会一直把自己“挂起”，直到得到这本书有没有的结果；如果是非阻塞式调用，你不管老板有没有告诉你，你自己先一边去玩了， 当然你也要偶尔过几分钟 check 一下老板有没有返回结果

注：套接字的默认状态是阻塞的（比如 tcp 的 read/recv/write/send 等操作），可设置为非阻塞。

2.3.4 linux I/O 多路复用(epoll/poll/select)

多路，是指多条连接；复用，是指多条连接复用一个阻塞对象；当多条线路共用一个阻塞对象后，进程只需要在一个阻塞对象上等待，而**无须再轮**

询所有连接，当某条连接有新的数据可以处理时，操作系统会通知进程，进程从阻塞状态返回，开始进行业务处理。

epoll 是 Linux 内核为处理大批量文件描述符而作了改进的 poll，是 Linux 下多路复用 IO 接口 select/poll 的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统 CPU 利用率。另一点原因就是获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核 IO 事件异步唤醒而加入 Ready 队列的描述符集合就行了。

注：select/poll 可查阅《unix 网络编程》卷 1 第六章

2.3.5 server 端网络模型

28. PPC：Proces per Connection，其含义是指每次有新的连接就新建一个进程去专门处理这个连接的请求。由于创建进程的开销较大性能不高。
29. TPC：Thread per Connection，其含义是指每次有新的连接就新建一个线程去专门处理这个连接的请求。由于创建线程的开销较大性能不高。
30. Reactor：一种设计模式，本质上就是多路复用+资源池（CPU 并发池），这里的资源池可以是进程池/线程池。Nginx 是 epoll+多进程模型，每个进程是单线程。Memcache 是 epoll+多线程模型。

注：在 linux 下用 c/c++分别实现源生 tcp server（socket/bind/listen/accept）和 epoll，有助于理解本节内容。

2.3.6 client 端网络模型

31. 以 Nginx 为例，作为 Web 服务器，Nginx 被动接受客户端主动发起的请求，同时，在有些请求的处理过程中，Nginx 也需要向其他上游服务器发起请求，比如我们常用的 proxy_pass+upstream 功能，这种请求称为主动连接。Nginx 在发起主动连接时，通常会维护一个连接池，而维护这个主动连接池的读写状态通知，在 linux 下常用也是 epoll 多路复用技术。
32. 目前酷狗 rpc 是基于 Nginx 实现的，rpc 作为一个客户端，用的就是上面介绍的网络模型。

注： 可阅读《深入理解 Nginx》第 8/9 章。

2.3.7 异步非阻塞网络调用举例

要达到高性能，在单线程/协程环境下，我们要尽力避免阻塞函数的调用。以下是常用的异步网络调用指引。

33. nginx+lua（openresty）

1.1 使用 ngx.location.capture()+proxy_pass+upstream 方式（首选）

1.2 基于 cosocket 封装的 http 库。（https://github.com/ledgetech/lua-resty-http）

1.3 其他方式不是首选，比如 luasocket 是阻塞的。

34. nginx+php-fpm

2.1 php-fpm 是多进程架构，一个 php 子进程在同一时刻只能处理一个请求，资源独占情况下不讨论异步非阻塞。

35. Python

3.1 http 请求，使用 gevent+monkey+requests

```
from gevent import monkey
import gevent
import requests
monkey.patch_all()
```

3.2 mysql 请求，使用 gevent+pymysql，（MySQL 库底层实现是阻塞的，不能用）

36. Go

4.1 http 请求，官网库已经支持异步网络请求，goroutines+http 库，

37. Java

5.1 Netty 库

2.4 磁盘延迟（戴育东）

38. 需要了解机械磁盘寻址与 SSD 寻址的区别，SSD 随机读写更快。

39. 会用 iostat 命令，关注磁盘的整体情况，理解%until 指标与%await 指标。

40. 会用 iotop 命令，比如查看哪个进程在消耗磁盘 IO（*iotop -oP*）。

41. 磁盘 IO 过大，会引起 CPU 负载飙升，需留意 top 命令的%wa 指标。

2.5 网络延迟（戴育东）

42. 会用 iftop 命令，观察网卡流量来源，比如观察网卡 eth1（**iftop -i eth1**）。
43. 会用 netstat 命令，观察连接情况，比如统计 TCP 80 端口（**netstat -natl | grep 80 | awk '{print $6}' | sort | uniq -c | sort -nr**）。
44. 当跨机房 ping 丢包率达 3%，持续 5 分钟，联系运维处理。
45. 当跨机房 ping 延时>100ms，持续 5 分钟，联系运维处理。
46. 机房网络告警可在 KMC 订阅。

3 安全（李子兴）

3.1 主机安全

3.1.1 公网端口

一个端口就是一个潜在的通信通道，也就是一个入侵通道。黑客对目标计算机进行端口扫描，能得到许多有用的信息，从而发现系统的安全漏洞。**不允许开发私自开放外网端口**，所有业务程序，只运行绑定本机或内网 IP 端口。运维有端口扫描工具，可以监控发现未允许开放的外网端口。

非法端口开发自查方法是，运行 netstat –antlp 查看下服务器是否有未被授权的端口被监听，查看下对应的 pid。检查服务器是否存在恶意进程,恶意进程往往会开启监听端口，与外部控制机器进行连接。

若发先有非授权进程，运行 ls -l /proc/$PID/exe 或 file /proc/$PID/exe（$PID 为对应的 pid 号），查看下 pid 所对应的进程文件路径。如果为恶意进

程，删除下对应的文件即可。

3.1.2 漏洞

通过端口扫描，可以发现开放的端口，如果这些端口上跑着的服务还有漏洞，攻击者可以通过这些漏洞入侵主机。主要应对措施有两个：1、应用最小化运行；2、及时升级修复应用程序漏洞。

1、应用最小化运行

如果您服务器内有运行 Web、数据库等应用服务，请限制应用程序账户对文件系统的写权限，同时尽量使用非 root 账户运行。使用非 root 账户运行可以保障在应用程序被攻陷后攻击者无法立即远程控制服务器，减少攻击损失。

样例：

- 进入 web 服务根目录或数据库配置目录；
- 运行 chown -R nginx:nginx /data1/logs/xxxx、chmod -R 750 file1.txt 命令配置网站访问权限。

2、升级修复应用程序漏洞

机器被入侵，部分原因是系统使用的应用程序软件版本较老，存在较多的漏洞而没有修复，导致可以被入侵利用。比较典型的漏洞如 ImageMagick、openssl、glibc 等，通过升级进行修复。

3.2 账号安全（李子兴）

3.2.1 鉴权

鉴权主要是指接口鉴权，包括终端接口鉴权及内外接口鉴权。

终端接口鉴权首先需走 opd 申请安全凭证 appid、appkey；内外接口鉴权则申请 serverid、serverkey。

然后计算出签名串，具体生成规则参考云计算 wiki 接口规范。

对于涉及用户权限接口，接口务必要要求传 token 参数，服务端校验。

接口鉴权失败后，应根据实际情况反回合理的错误码及错误信息。

3.2.2 协议加密

为了防止数据传输过程中被篡改，通过 https 方式或在有选择地通过 rsa+aes 方式加密数据。

其中，token 不允许明文传输。

3.2.3 账号防盗

不允许客户端记录用户密码，禁止使用密码登录方式实现自动登录，token 需设置有效期，各 app 间的 token 权限隔离。

在 token 未授权情况，接口不允许下发敏感数据，包括但不限于：手机号、用户名、邮箱、密码、音乐包有效期等。

用户进行交易、改个人数据、UGC、社交（看空间、访客、查用户资料也算）时，必须校验用户令牌（token）。

3.3 业务安全（李子兴）

3.3.1 DDOS/CC 攻击

DDOS 只能缓解而不能完全防御，其核心是保证服务于业务的带宽不被打满，防御的手段通常分下面 4 层：

第一层，ISP 近源清洗。（需要 ISP 做，目前没见过哪个 ISP 实现）

第二层，云清洗/CDN 硬抗。（**最佳防御：打散接入 IP，局部 IP 对用户可见。**CDN 就是这样）

第三次，DC 层近目的清洗。（为时已晚，较难避免误杀）

第四层，OS/APP 层抗 CC。（较难避免误杀，但可作为最后的降级）

更多可参考：http://www.freebuf.com/articles/network/78660.html

3.3.2 业务防刷

业务防刷通过接入风控系统达到保护业务的目的。UGC 类业务，如：登录、注册、关注、评论、点赞、投票、及付费业务等，要求都接入风控系统。

接口要求遵守云技术接口规范，必传参数及标准错误码、错误信息必须要有。

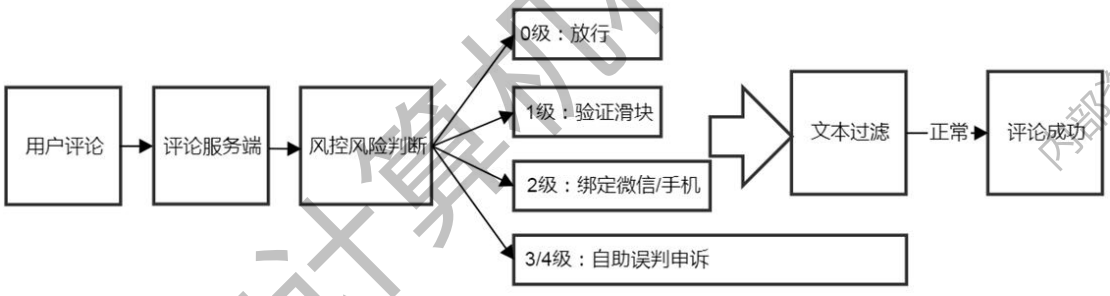
风控系统主要通过手机、QQ、微信、账号、设备、IP 等等维度建模，使用大数据分析其恶意程度。如果接口缺少这些必备字段，则防刷的效果就要大打折扣。

业务防刷的指导原则，是**使黑产的收益小于安全对抗成本**。因此发放福利的接口，一定要经过安全架构师的评审，评估好黑产收益，对应做好对抗措施。比

如福利发 3￥每账号，则可以挑选账号价值超过 3￥的用户群体来发放，以免被黑产撸羊毛。

3.3.3 规范

47. **业务防刷接入风控（对接向燕军）。**IP 黑名单只在被刷挂的紧急时候用，有效期不能超过三天。
48. **禁止共用 businessid。**一个业务分配一个 businessid，不同 businessid 对应风控引擎规则不一样。
49. 按类别应用风控措施。福利类，如抽奖，风控提供用户价值，**业务发放的总福利不应大于用户价值。**非福利类，如评论，**前端必须接入风控系统提供的安全验证模块**，让误伤用户有路可走。流程参考：



50. **端到端监控。**要求业务把风控的执行结果回传给风控系统。

3.4 图片文本安全（李子兴）

51. 图片，头像、相册、背景图等图片业务必须接入图片审核审核服务（对接林泽浜）
52. 文本，IM、聊天、评论、弹幕、留言、昵称、签名、论坛、问答等言论业务必须接入敏感词过滤接口（对接黄健）。

3.5 数据安全

数据安全首先是要防用户隐私泄露。这不仅要求我们做好鉴权和通讯加密，也要防拖库。

防拖库的首要措施是对数据进行加密存储，密钥由通用加密服务（向炼）负责管理。

4 设计

4.0 综述（梁海龙）

设计的本质是**高聚合低耦合**。

设计的要点是**封装与接口**。实现这两要点有各种形式，函数、类、闭包、API、协议、消息等等。

设计的质量，取决于工程师是否**意识**到问题的紧迫性。优秀的设计师不仅仅解决当下的问题，还能解决未来一段时间的问题。

设计的评判标准是是否**适度**。**缺乏设计**导致大泥团。**过度设计**则不仅浪费人力，往往还难以理解。

设计的演进，首先就是**拆**。**垂直拆分**业务和流程，**深度拆分**调用层次。

当你一个函数里面的意大利面条式代码超过 200 行的时候，就是时候思考一下，这个大流程是否要分拆出来一些可聚合的小流程？或者里面的业务逻辑是否要和数据存取操作分层？

当你的系统过于庞大和复杂的时候，就时候考虑一下，你这个系统是否承当了过多的职责？是不是该垂直上划分几个模块了？或者是不是该深度拆分为应用层

和基础服务层了?

设计的演进，其次就是**合。**当拆分过度，或者为了快速迭代产生了雷同的多套服务时，运营和维护成本成为了难题，这是就要考虑**平台化、云化。**提供统一平台对接多个业务。

4.1 代码模块（向炼）

53. **SOLID 面向对象设计原则**

Ø 单一职责原则 (The Single Responsibility Principle)

修改某个模块的理由应该只有一个，如果超过一个，说明模块承担不止一个职责，要视情况拆分。

Ø 开放封闭原则 (The Open Closed Principle)

软件实体应该对扩展开放，对修改封闭。一般不要直接修改类库源码（即使你有源代码），通过继承等方式扩展。

Ø 里氏替换原则(The Liskov Substitution Principle)

当一个子类的实例能够被替换成任何超类的实例时，它们之间才是真正的 is-a 关系。

Ø 接口隔离原则(The Interface Segregation Principle)

接口隔离原则的含义是：建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。

采用接口隔离原则对接口进行约束时，要注意以下几点：

接口尽量小，但是要有限度。对接口进行细化可以提高程序设计灵活性是不争的事实，但是如果过小，则会造成接口数量过多，使设计复杂化。所以一定要适度。

为依赖接口的类定制服务，只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。

Ø 依赖倒置原则(Dependence Inversion Principle - DIP)

高层模块不应该依赖于底层模块的实现细节，应该依赖于抽象接口。换句话说，依赖于抽象，不要依赖于具体实现。比方说，你不会把电器电源线焊死在室内电源接口处，而是用标准的插头插在标准的插座（抽象）上。

54. **AKF 架构原则**

Ø N + 1 设计

永远不要少于两个，通常为三个。比方说无状态的 Web/API 一般部署至少>=2个。

Ø 回滚设计

确保系统可以回滚到以前发布过的任何版本。可以通过发布系统保留历史版本，或者代码中引入动态开关切换机制（Feature Switch)。

Ø 禁用设计

能够关闭任何发布的功能。新功能隐藏在动态开关机制（Feature Switch）后面，可以按需一键打开，如发现问题随时关闭禁用。

Ø 监控设计

在设计阶段就必须考虑监控，而不是在实施完毕之后补充。例如在需求阶段就要考虑关键指标监控项，这就是度量驱动开发（Metrics Driven Development)的理念。

Ø 使用成熟的技术

只用确实好用的技术。商业组织毕竟不是研究机构，技术要落地实用，成熟的技术一般坑都被踩平了，新技术在完全成熟前一般需要踩坑躺坑。

Ø 异步设计

能异步尽量用异步，只有当绝对必要或者无法异步时，才使用同步调用。

Ø 无状态系统

尽可能无状态，只有当业务确实需要，才使用状态。无状态系统易于扩展，有状态系统不易扩展且状态复杂时更易出错。

Ø 水平扩展而非垂直升级

永远不要依赖更大、更快的系统。一般公司成长到一定阶段普遍经历过买更大、更快系统的阶段。

Ø 设计时至少要有两步前瞻性

在扩展性问题发生前考虑好下一步的行动计划。

Ø 小构建、小发布和快试错

全部研发要小构建，不断迭代，让系统不断成长。这个和微服务理念一致。

Ø 隔离故障

实现故障隔离设计，通过断路保护避免故障传播和交叉影响。通过舱壁泳道等机制隔离失败单元（Failure Unit)，一个单元的失败不至影响其它单元的正常工作。

Ø 自动化

设计和构建自动化的过程。如果机器可以做，就不要依赖于人。自动化是 DevOps 的基础。

55. **其他原则**

Ø 最少知识原则(Least Knowledge Principle - LKP)

最少知识原则又叫迪米特法则。核心思想是：低耦合、高内聚

一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。也就是说一个软件实体应当尽可能少的与其他实体发生相互作用。这样，当一个模块修改时，就会尽量少的影响其他的模块，扩展会相对容易，这是对软件实体之间通信的限制，它要求限制软件实体之间通信的宽度和深度。

Ø 信息专家（Information Expert)

为对象分配职责的通用原则 – 把职责分配给拥有足够信息可以履行职责的专家。

Ø 好莱坞原则

好莱坞明星的经纪人一般都很忙，他们不想被打扰，往往会说：Don't call me, I'll call you. 翻译为：不要联系我，我会联系你。对应于软件设计而言，最著名的就是“控制反转”（或称为“依赖注入”），我们不需要在代码中主动的创建对象，而是由容器帮我们来创建并管理这些对象。

Ø 组合/聚合复用原则（Composition/Aggregation Reuse Principle - CARP）

当要扩展类的功能时，优先考虑使用组合，而不是继承。这条原则在 23 种经典设计模式中频繁使用，如：代理模式、装饰模式、适配器模式等。

Ø 不要重复你自己（Don't repeat yourself - DRY）

不要让重复的代码到处都是，要让它们足够的重用，所以要尽可能地封装。

Ø 保持它简单与傻瓜（Keep it simple and stupid - KISS）

不要让系统变得复杂，界面简洁，功能实用，操作方便，要让它足够的简单，足够的傻瓜。

Ø 惯例优于配置（Convention over Configuration - COC）

尽量让惯例来减少配置，这样才能提高开发效率，尽量做到“零配置”。很多开

发框架都是这样做的

Ø 命令查询分离（Command Query Separation - CQS）

在定义接口时，要做到哪些是命令，哪些是查询，要将它们分离，而不要揉到一起。

Ø 关注点分离（Separation of Concerns - SOC）

将一个复杂的问题分离为多个简单的问题，然后逐个解决这些简单的问题，那么这个复杂的问题就解决了。难就难在如何进行分离。

Ø 你不需要它（You aren't gonna need it - YAGNI）

不要一开始就把系统设计得非常复杂，不要陷入“过度设计”的深渊。应该让系统足够的简单，而却又不失扩展性，这是其中的难点。

4.2 可扩展（李长豪）

可扩展是指对现有系统影响最小的情况下，系统功能可持续扩展或提升的能力。

可扩展性是一种设计概念，我们希望在现有的架构或者设计基础上，当未来某些方面发生改变的时候，能够以最小的改动来适应这种变化。我们的改动越小，并且对这种变化的适应性越好，我们就说这个设计的可扩展性是越好的。简单来说，就是让当前设计去适应未来不确定的变化。

4.2.1 开闭原则

开闭原则是指一个软件实体应当对扩展开放，对修改关闭。这一原则最早由Brtrand Meyer 提出，英文原文是

Software entities should be open for extension, but closed for modification

这个原则说的是，在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展。换言之，应当可以在不必修改源代码的情况下改变这个模块的行为。

所有的软件系统都有一个共同的性质，即对他们的需求都会随时间的推移而发生变化。在软件系统面临新的需求时，系统的设计必须是稳定的。满足开闭原则的设计可以给一个软件系统两个无可比拟的优越性。

- 通过扩展已有的软件系统，可以提供新的行为，以满足对软件的新需求，使变化中的软件系统有一定的适应性和灵活性。
- 已有的软件模块，特别是最重要的抽象层模块不能再修改，这就使变化中的软件系统有一定的稳定性和延续性。

4.2.2 抽象化是关键

不能修改而可以扩展，似乎是自相矛盾的。怎么可以同时又不修改，而又可扩展呢?

解决问题的关键在于抽象化。在支持面向对象的编程语言里，可以定义一个或多个抽象类或接口，规定出所有的具体类必须提供的方法的特征（Signature）作为系统设计的抽象层。这个抽象层预见了所有的可能扩展，因此，在任何扩展情况下都不会改变。这就使得系统的抽象层不需修改。

同时，由于从抽象层导出一个或多个新的具体类可以改变系统的行为，因此系统的设计对扩展是开放的。

4.2.3 协议规范

好的协议规范应该可扩展，可以使用 json、PB、msgpack，也可以使用 PHP、lua、C++、java 方便的编解码。

1．JSON

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。易于人阅读和编写。同时也易于机器解析和生成。它基于 JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999 的一个子集。JSON 采用完全独立于语言的文本格式，但是也使用了类似于 C 语言家族的习惯（包括 C, C++, C#, Java, JavaScript, Perl, Python 等）。这些特性使 JSON 成为理想的数据交换语言。

JSON 建构于两种结构

56. “名称/值”对的集合（A collection of name/value pairs）。不同的语言中，它被理解为对象（object），纪录（record），结构（struct），字典（dictionary），哈希表（hash table），有键列表（keyed list），或者关联数组（associative array）。
57. 值的有序列表（An ordered list of values）。在大部分语言中，它被理解为数组（array）。

这些都是常见的数据结构。事实上大部分现代计算机语言都以某种形式支持它们。这使得一种数据格式在同样基于这些结构的编程语言之间交换成为可能。

value 字段不能直接保存二进制数据

有业务在 json 的 value 里面保存了二进制，在 openresty 能正常编解码(openresty 做了适配)，但是换了别的语言会解析失败。比如 go 自带的 json 包，c 语言的开源 cjson。 如果要在 value 里面保存二进制，可以先把内容进行 base64 编码。

2．Protocol Buffer

protobuf(Google Protocol Buffers)是 Google 提供一个具有高效的协议数据交换格式工具库(类似 Json)，但相比于 Json，Protobuf 有更高的转化效率，时间效率和空间效率都是 JSON 的 3-5 倍。

Protobuf 提供了 C++、java、python 语言的支持，提供了 windows(proto.exe)和 linux 平台动态编译生成 proto 文件对应的源文件。proto 文件定义了协议数据中的实体结构(message ,field)

关键字 message: 代表了实体结构，由多个消息字段(field)组成。

消息字段(field)：包括数据类型、字段名、字段规则、字段唯一标识、默认值

数据类型：常见的原子类型都支持

字段规则：

58. required：必须初始化字段，如果没有赋值，在数据序列化时会抛出异常

59. optional：可选字段，可以不必初始化。

60. repeated：数据可以重复(相当于 java 中的 Array 或 List)

61. 字段唯一标识：序列化和反序列化将会使用到。

默认值：在定义消息字段时可以给出默认值。

3．MsgPack

MessagePack 是一个高效的二进制序列化格式。它让你像 JSON 一样可以在各种语言之间交换数据。但是它比 JSON 更快、更小。小的整数会被编码成一个字节，短的字符串仅仅只需要比它的长度多一字节的大小。

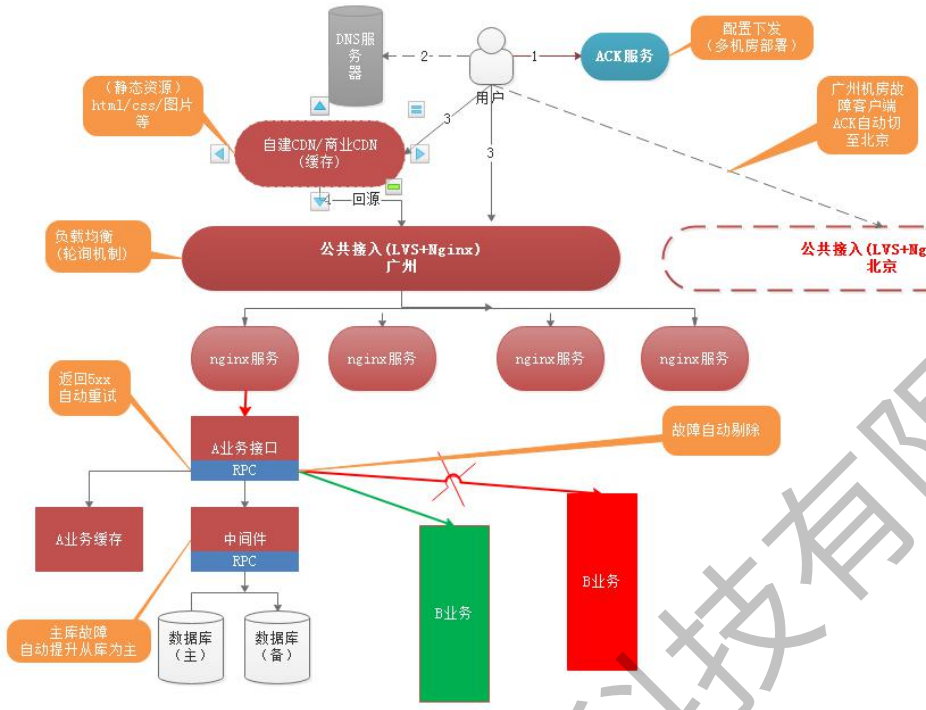
官方用一句话总结了这个东东：

It’s like JSON.but fast and small.

核心压缩方式可参看官方说明 messagepack specification，概括来讲就是:

- true、false 之类的：直接给 1 个字节，（0xc3 表示 true，0xc2 表示 false）
- 不用表示长度的：就是数字之类的，他们天然是定长的，是用一个字节表示后面的内容是什么，比如用（0xcc 表示这后面，是个 uint 8，用 oxcd 表示后面是个 uint 16，用 0xca 表示后面的是个 float 32)。对于数字做了进一步的压缩处理，根据大小选择用更少的字节进行存储，比如一个长度<256 的 int，完全可以用一个字节表示。
- 不定长的：比如字符串、数组、二进制数据（bin 类型），类型后面加 1~4 个字节，用来存字符串的长度，如果是字符串长度是 256 以内的，只需要 1 个字节，MessagePack 能存的最长的字符串，是(2^32 -1 ）最长的 4G 的字符串大小。
- 高级结构：MAP 结构，就是 k-v 结构的数据，和数组差不多，加 1~4 个字节表示后面有多少个项
- Ext 结构：表示特定的小单元数据。也就是用户自定义数据结构。

4.3 线上架构（杨权新）



4.3.1 接入层：失败、慢、劫持（杨权新）

1. 自建 CDN:

自建 cdn 的容灾策略做的会比较好一些，但是缓存空间比较小，带宽有限。不适合做命中率比较低、或者存储量比较大，比如超过 10G 的存储的业务。

2. 商用 CDN：

覆盖点多，缓存空间和带宽大

具体接入参考：http://kcr.opd.kugou.net/w/ops/work/cdn/applyforcdn/

3. DNS：GSLB

实现在广域网（包括互联网）上不同地域的服务器间的流量调配，保证使用最佳的服务器服务离自己最近的客户，从而确保访问质量。灵活策略下发：地区和运营商。

4. ACK，容灾网关

Ack 支持多种容灾策略：多域名、域名优先解析和容灾网关，配置下发支持包括：客户端版本号、地区、运营商。

多域名，利用 dns 功能保证用户就近访问，主域名失败可以重试备用域名。

域名优先解析，用户不保证就近访问，可以解决域名劫持问题。

容灾网关，ip 直连访问服务，接入点少且部署边缘机房，业务容灾最后通道。

协议支持：https、http2、quic、http。

62. **https**：数据加密安全传输，可以防止域名和数据劫持导致业务失败，底层网络传输是 TCP。
63. **Quic：**数据加密安全传输，可以防止域名和数据劫持导致业务失败，底层网络传输是 UDP，网络切换（4G->3G 等）不需要重新 SSL 握手，弱网环境有优化，缺点是部分运营商会屏蔽 UDP 协议导致网络不可达。
64. **http2：**数据非安全传输，多路复用可以提升业务并发请求性能，使用 web 页面，接口优于并发场景不多优势不明显，缺点默认连接是长连接，若连接不可用需要应用显式发起 ping 帧才会重连。
65. **http：**数据非安全传输，请求-响应串行处理，web 页面类性能比 http2 差。

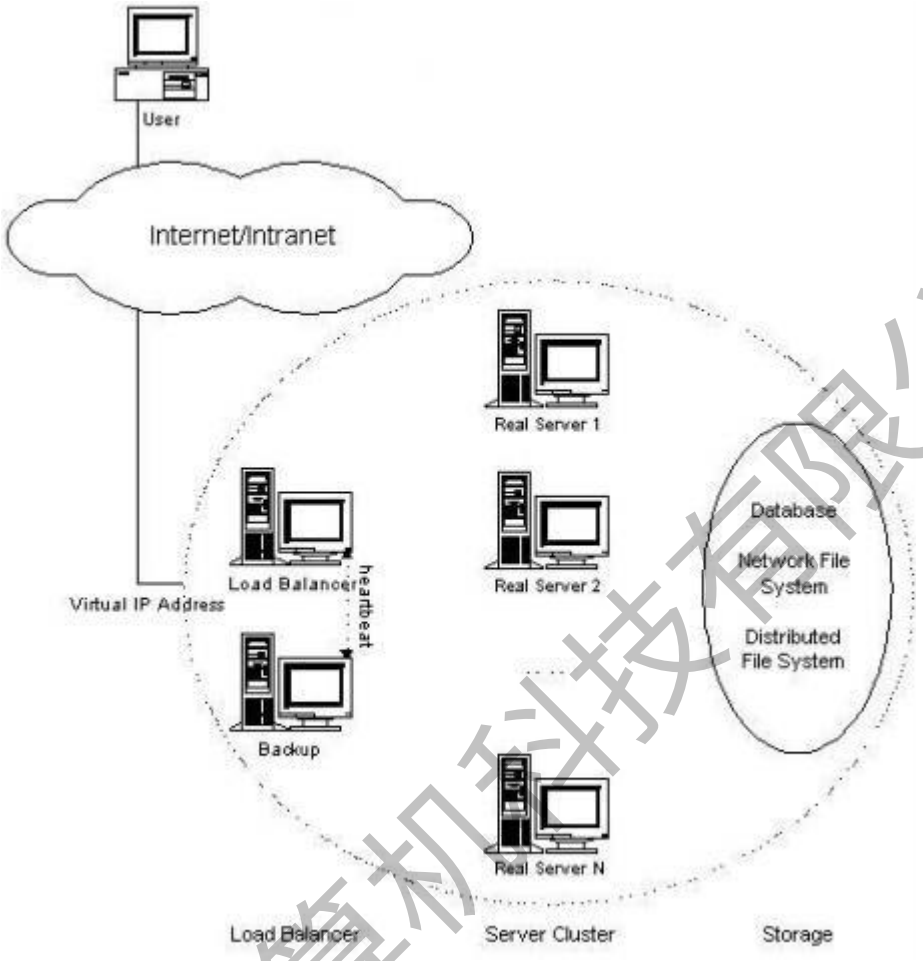
5. 公共接入：OSPF、LVS、Nginx

66. **OSPF(开放式最短路径优先)**：是一个内部网关协议(Interior Gateway Protocol，简称 IGP)，用于在单一自治系统（autonomous system,AS）内决策路由。是对链路状态路由协议的一种实现，隶属内部网关协议 (IGP)，故运作于自治系统内部。著名的迪克斯加算法(Dijkstra)被用来计算最短路径

树。

67. **LVS:** 是 Linux Virtual Server 的简写，意即 Linux 虚拟服务器，是一个虚拟的服务器集群系统，用于实现负载平衡，三种负载均衡方式：NAT、TUN 和 DR
68. **NAT**：支持端口映射，RS 可以使用任意操作系统，节省公有 IP 地址，请求和响应报文都要经过 Director 转发;极高负载时，Director 可能成为系统瓶颈。
69. **TUN**：基于隧道封装技术。在 IP 报文的外面再包一层 IP 报文。当 Director 接收到请求的时候，选举出调度的 RealServer 当接受到从 Director 而来的请求时，RealServer 则会使用 lo 接口上的 VIP 直接响应 CIP。这样 CIP 请求 VIP 的资源，收到的也是 VIP 响应。优点是 RIP,VIP,DIP 都应该使用公网地址，且 RS 网关不指向 DIP，只接受进站请求，解决了 LVS-NAT 时的问题，减少负载。请求报文经由 Director 调度，但是响应报文不需经由 Director。；缺点是不指向 Director 所以不支持端口映射，RS 的 OS 必须支持隧道功能，隧道技术会额外花费性能，增大开销。
70. **DR**：当 Director 接收到请求之后，通过调度方法选举出 RealServer，将目标地址的 MAC 地址改为 RealServer 的 MAC 地址，RealServer 接受到转发而来的请求，发现目标地址是 VIP，RealServer 配置在 lo 接口上，处理请求之后则使用 lo 接口上的 VIP 响应 CIP。优点是 RIP 可以使用私有地址，也可以使用公网地址，只要求 DIP 和 RIP 的地址在同一个网段内，请求报文经由 Director 调度，但是响应报文不经由 Director，RS 可以使用大多数 OS；缺点是不支持端口映射，不能跨局域网。

LVS 集群采用三层结构，其体系结构如下：



4.3.2 业务层：研发效率、性能、异步化（杨权新）

Openresty 基于 Nginx 提供高效异步 IO，中国人章亦春把 LuaJIT VM 嵌入到 Nginx 中，实现了 OpenResty 这个高性能服务端解决方案。开发门槛低，这一切都是用强大轻巧的 Lua 语言来操控，Openresty 为之提供了动态语言的灵活，当性能与灵活走在了一起，无疑对于被之前陷于臃肿架构，苦于提升性能的工程师来说是重大的利好消息，适用于接口开发，不利于在大文件传输或者 CPU 密集型业务场景，缺点类库比较少。

PHP 开发速度快，部署、维护极其简单，在 web 领域，php 确实是一门开箱即可使用的语言，类库丰富，缺点请求同步处理（非 swool 框架）不能满足高并发业务场景。

C++是一门面向对象编译型语言，入门门槛比较高，类库丰富，高性能和异步化框架 IO 难 hou 住，适合基础框架平台类业务场景，对操作系统和操作系统底层有比较了解的高级开发人员。

JAVA 是一门面向对象语言，类库丰富，开发人员众多，易招聘，即适用于大数据处理，也适用业务开发，不需要过多担心内存泄漏，社区丰富，开源框架满天飞，开发人员很多问题都可以轻松解决。

GO 是一门比较流行语言，类库和开源框架丰富，执行文件不需要外部依赖，可以利用协程技术解决业务高性能和异步化，依托现在最流行的云计算，社区推广很不错，入门简单，适用于接口、基础框架和组件类开发，不适合 web 开发。

4.3.3 调用层：失败、慢（杨权新）

RPC 解决内部服务之间调用去单点问题，业务接入简单，业务故障自动转移，容灾策略丰富、灵活。

具体可以参考：http://git.kugou.net/kugou-rpc/document

4.3.4 数据层：IO 性能、正确性、数据一致性（杨权新）

主从、主主、读写分离

MySQL 主从同步策略：全同步，半同步和异步复制

全同步复制，当主库提交事务之后，所有的从库节点必须收到、APPLY 并

且提交这些事务，然后主库线程才能继续做后续操作。但缺点是，主库完成一个事务的时间会被拉长，性能降低。

半同步复制，是介于全同步复制与全异步复制之间的一种，主库只需要等待至少一个从库节点收到并且 Flush Binlog 到 Relay Log 文件即可，主库不需要等待所有从库给主库反馈。同时，这里只是一个收到的反馈，而不是已经完全完成并且提交的反馈，如此，节省了很多时间。

异步复制，主库将事务 Binlog 事件写入到 Binlog 文件中，此时主库只会通知一下 Dump 线程发送这些新的 Binlog，然后主库就会继续处理提交操作，而此时不会保证这些 Binlog 传到任何一个从库节点上。

MySQL 主主模式：指双活互为备份，异步复制同样存在丢失数据的风险，而且要解决数据的冲突问题，解决冲突最好的方法是避免冲突，比如自增 ID 奇偶方案，该模式慎用，主键冲突将导致数据无法同步。

MySQL 读写分离：解决数据库写入，影响查询效率；针对非强一致性业务场景

CAP 原理：指的是在一个分布式系统中，Consistency（一致性）、Availability（可用性）、Partition tolerance（分区容错性），三者不可得兼。一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份最新的数据副本）可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备高可用性）分区容错性（P）：以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在 C 和 A 之间做出选择。

Raft 一致性算法:

CAP 定理，指的是在一个分布式系统中，Consistency（一致性）、Availability（可用性）、Partition tolerance（分区容错性），三者不可得兼。一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份最新的数据副本）可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备高可用性）分区容错性（P）：以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在 C 和 A 之间做出选择。

Paxos 和 Raft 都是数据强一致性，性能不高，NRW 如果 N=3,R=2,W=2（满足 R + W > N）属于数据强一致性，若满足 R + W <= N 系统只能保证最终一致性，强一致性牺牲行为为代价。

TCC 为了解决在事务运行过程中大颗粒度资源锁定的问题，业界提出一种新的事务模型，它 是基于业务层面的事务定义。锁粒度完全由业务自己控制。它本质是一种补偿的思路。它把事务运行过程分成 Try、Confirm / Cancel 两个阶段。在每个阶段的逻辑由业务代码控制。这样就事务的锁粒度可以完全自由控制。业务可以在牺牲隔离性的情况下，获取更高的性能。

· Try 阶段

o Try ： 尝试执行业务

完成所有业务检查(一致性)

预留必须业务资源(准隔离性)

· Confirm / Cancel 阶段:

o Confirm ： 确认执行业务

真正执行业务

不做任务业务检查

Confirm 操作满足幂等性

o Cancel ： 取消执行业务

释放 Try 阶段预留的业务资源

Cancel 操作满足幂等性

o Confirm 与 Cancel 互斥

Gossip 算法又被称为反熵（Anti-Entropy），熵是物理学上的一个概念，代表杂乱无章，而反熵就是在杂乱无章中寻求一致，这充分说明了 Gossip 的特点：在一个有界网络中，每个节点都随机地与其他节点通信，经过一番杂乱无章的通信，最终所有节点的状态都会达成一致。每个节点可能知道所有其他节点，也可能仅知道几个邻居节点，只要这些节可以通过网络连通，最终他们的状态都是一致的，当然这也是疫情传播的特点。简单的描述下这个协议，首先要传播谣言就要有种子节点。种子节点每秒都会随机向其他节点发送自己所拥有的节点列表，以及需要传播的消息。任何新加入的节点，就在这种传播方式下很快地被全网所知道。这个协议的神奇就在于它从设计开始就没想到信息一定要传递给所有的节点，但是随着时间的增长，在最终的某一时刻，全网会得到相同的信息。当然这个时刻可能仅仅存在于理论，永远不可达。典型的应用场景如 redis 集群，无中心架构。

数据库选型表

4.4 生产流程（李毅）

4.4.1 需求

71. **优先级**
72. 为了确保将有限的资源投放在最有价值的地方，我们要求所有需求都要有合理的优先级，开发测试需要严格按照任务的优先级进行工作安排。
73. 优先级是以项目的重要程度、可带来的收益、或正在发生的影响等因素综合评估。优先级排序（从高到低）：1 -> 蓝旗 -> 黄旗 -> 2 -> 3。
74. 优先级 1：随客户端发版本的核心任务，团长控制。
75. 蓝旗：云技术部核心服务技术需求、重要项目的立项任务、线上问题修复任务、安全整改任务等，由总监梁海龙评估插旗。
76. 黄旗：云技术部重要的技术改进需求，各开发组长认为对业务比较重要的任务，由云技术部各开发组长评估插旗。
77. 优先级 2：排在黄旗之后的普通任务，由需求方自己定义。
78. 优先级 3：排在优先级 2 任务之后的普通任务，是需求的默认优先级，由需求方自己定义。
79. 特殊说明：当有多个同一优先级的任务并行时，当事人应主动找各需求方进行优先级 PK，确定那个任务更优先，必要时可以找上级参与评估。
80. **需求质量**

需求的质量直接影响到研发质量、研发效率、产品质量，所以我们必须要对需求质量做出一定的要求：

1）为了防止研发过程中对开发、测试和产品对需求有不一致的理解，要求需求必须清晰明确，需求的描述以第三方（或者自己的上级）能看懂为标准。

2）同一需求对应多个开发任务时，需要在需求描述中根据任务的拆分，明确每个任务的需求，用任务 ID 分隔。

3）在需求评审、开发、测试阶段，对需求的理解及沟通结果要及时同步更新到 MTP。

4）开发、测试不应该盲目接受自己不确定或者没有清楚理解的需求，应主动找需求方沟通，将疑点明确后再开始执行。

81. **方案质量**

方案质量决定了产品的稳定性、性能、研发效率及后期的维护成本，我们要求所有公用类库的技术选型、基础架构设计、性能调优、框架使用、复杂数据存储、缓存机制、容灾策略、安全防范、复杂的产品逻辑实现……等技术、产品方案在实施前，必须通过架构师的方案评审。

4.4.2 持续交付

持续交付当前是一种非常流行的软件开发策略，用于优化软件交付流程，以尽快得到高质量、有价值的软件。这种方法可以让你能更快地验证业务想法，通过直接在用户那里进行试验，做到快速迭代。

持续交付是在持续集成的基础上，将集成后的代码部署到更贴近真实运行环境的「类生产环境」（production-like environments）中。持续交付优先于整个产品生命周期的软件部署，建立在高水平自动化持续集成之上。

持续交付有以下优点：

1）快速发布。能够应对业务需求，并更快地实现软件价值。

2）编码->测试->上线->交付的频繁迭代周期缩短，同时获得迅速反馈；

3）高质量的软件发布标准。整个交付过程标准化、可重复、可靠，整个交付过程进度可视化，方便团队人员了解项目成熟度；

4）更先进的团队协作方式。从需求分析、产品的用户体验到交互 设计、开发、测试、运维等角色密切协作，相比于传统的瀑布式软件团队，更少浪费。

对于持续交付，我们团队该怎么做?

我们团队应按照前面小步迭代的原则将任务拆分成可独立上线的特性，完成任务的开发测试后，及时部署到预发布环境，如果预发布环境测试通过，就可以直接部署到线上生产环境，实现小步迭代和持续交付，这样可以有效的缩短产品研发周期，提高产品满意度，尽快将产品的新特性推向用户。当然，这也只是最初级的持续交付，真正意义上的持续交付需要有很完善的持续集成和自动化测试做支撑。